University of BRISTOL

Team B - Team Steam

BirdGang

Authors:
Team Manager: Jaya Sekhon
Lead Programmer: Connor Hamilton
Lead Design: Tanvi Pandey
Ambika Agarwal
Iban Hossain
Joe Dight
Emily Lopez Burst

May 6, 2022

1 Signed Contributions

1.1 Weightings

Team member	Weighting
Jaya Sekhon	1
Connor Hamilton	1
Tanvi Pandey	1
Ambika Agarwal	1
Iban Hossain	1
Joe Dight	1
Emily Lopez Burst	1

Signed:

J. SEKHON

C. HAMILTON

T. PANDEY

A. AGARWAL

I. HOSSAIN

J. DIGHT

E. LOPEZ BURST

2 Top Five Contributions

Team Video Link

https://www.youtube.com/watch?v=aOmKyhrso0Q

Contributions

1. Developed an intuitive and easy to use control scheme for flight. Our game controls have been carefully chosen to provide the user with seamless interaction with their bird character. This utilises Unity's physics engine to create fluid movement.
2. Made use of the Emmy-award winning Cinemachine tools to provide cinematic quality camera movement.
3. Implemented advanced AI systems. Firstly, a Boids algorithm to produce realistic flocking behaviours with additional custom behaviours such as boundary avoidance, goal targeting, and obstacle avoidance. Secondly, a networking solution to synchronise the position of over 200 agents.
4. Created 52 unique models using Maya and built 5 shaders, resulting in a cohesive theme throughout our game, bringing it to life.
5. Demonstrated exceptional project management and organisational skills. We wrote comprehensive meeting notes and followed a regular schedule for assigning and combining work, allowing us to manage seven people's work successfully.

3 Nine Aspects

3.1 Team Process

- Practiced a culture of open and honest communication, following the Netflix 4A's feedback guide (see Section 5.1)
- Set up a Microsoft Teams channel calendar and added our other commitments to make the organisation of group meetings easier (see Section 5.1.2)
- Produced comprehensive meeting minutes, agendas and managed task allocation (see Section 5.2)
- Used Kanban and Figma boards to make both short and long-term plans for the project (see Section 5.2)
- Held regular review meetings on a Monday afternoon and integration meetings on a Friday afternoon, to ensure the groups work remained cohesive and feedback could be collected (see Section 5.2)

3.2 Technical Understanding

- Extensively prototyped a variety of control schemes for flight, utilising different features of Unity's physics engine (see Section 8.2.1)
- Researched and developed AI agents that sync smoothly across a network (see Section 8.5)
- Researched, implemented and extended a complex flocking algorithm, which included obstacle avoidance (see Section 8.6)
- Researched the tools provided by Unity's Emmy-award winning Cinemachine and selected the ones best suited to our game (see Section 8.7)
- Made use of the Unity Profiler to pinpoint where optimisations were required to improve performance (see Section 8.11)

3.3 Flagship Technologies

- Intuitive physics-based flight and navigation (see Section 8.2)
- Dynamic physics-based environmental effects (see Section 8.2.3)
- Spring simulation and real-time rope physics (see Section 8.4)
- Large-scale crowd simulation, accurately synced over the network (see Section 8.5)
- Goal-based AI agents, with a custom built fleeing mechanism (see Section 8.5)
- Flocking algorithm extended with additional features, including obstacle avoidance (see Section 8.6)
- Custom built in-game animations, using both Unity and Maya (see Section 8.8.2)
- Dynamic SFX and particle effects (see Section 8.9 and Section 8.8.5)

3.4 Implementation & Software

- Produced a custom spawning system that allows us to manage the number of people and spawning biases (see Section 7.5)
- Made a game events manager to allow for structured gameplay across a network, that is easily extensible (see Section 8.1)
- Implemented local calculations for AI agent movement to reduce network traffic (see Section 8.5)
- Implemented a dynamic lighting system that gradually changes the scene from daytime to evening (see Section 8.8.7)
- Wrote and recorded custom audio cue to guide players through the game (see Section 8.9)
- Baked occlusion culling expanding upon default frustrum culling (see Section 8.11.1)
- Used two custom flocking algorithms to find a solution that accounts for both obstacle avoidance requirements and the visual aesthetic of flocks in the sky (see Section 8.11.6)

3.5 Tools, Development & Testing

- Held a meeting every Friday to integrate all group members' work since Monday (see Section 5.2)
- Practiced the Scrum model of agile development (see Section 5.2)
- Used a separate development branch to combine everyone's work and conduct in-house testing before pushing to main and deploying to WebGL (see Section 5.3)
- Conducted periodic player testing, making note of the commit number and using a pre-set questionnaire to provide consistency (see Section 5.4)
- Added Continuous Integration (CI) testing for all commits to the main branch (see Section 7.3)

- Added Continuous Deployment building and deploying our game to GitHub pages (see Section 7.3
- Implemented unit tests in Unity (see Section 7.4)
- Made use of prefabs to ensure any world changes could be done efficiently and modified a script to automate the transformation of any existing non-prefab objects into prefabs (see Section 8.8.6)
- Implemented Level of Detail on our most vertex-intensive assets to improve performance (see Section 8.11.3)

3.6 Game Playability

- Detailed tutorial that incrementally activates controls to provide players with a clear introduction to the core controls (see Section 4 and demo)
- Combined the use of the keyboard and the mouse/trackpad to create a seamless interaction between the bird character and the player (see Section 8.2.1)
- Cinematic cutscenes engage players with a dramatic introduction at the start of a new round and establish the location of the event in the city (see Section 8.7)
- Player indicators provide clear real-time locations of the player's teammates (see Section 8.10)
- Inclusion of an easily accessible UI menu, available throughout the game, that provides players with a recap of the controls and audio settings (see Section 8.8.8)
- Inclusion of a Heads Up Display UI element that indicates the progression of the game through the rounds and the time remaining (see Section 8.8.8)
- Waypoints allow for players to attract their teammates attention to a specific location in the city (see Section 8.10)

3.7 Look & Feel

- Intuitive and immersive flight experience (see Section 8.2)
- Creation of a mood board that includes inspiration images for different parts of the city and other aspects, such as character models (see Section 8.8.1)
- Created five custom shaders to add more depth and detail to our world (see Section 8.8.4)
- Effective use of Unity's versatile particle system to enhance the users experience of gameplay (see Section 8.8.5)
- Cohesive art style, theme, and branding throughout the entire system (see Section 8.8)
- Fully custom 2D image artwork throughout the game (see Section 8.8.8)

3.8 Novelty & Uniqueness

- Completely novel game idea and playful, lighthearted world (see Section 4)
- Strong narrative running through the game (see Section 4)
- AI based agents, with entertaining fleeing behaviour (see Section 8.5)
- In-house production of 52 unique and cohesive models (see Section 8.8.2)
- Open world game (see demo)

3.9 Report & Documentation

- Recorded high-quality voiceovers, with access to a university microphone (see Section 8.9 and demo)
- Combined the expertise of the entire group to create a detailed report that contains a comprehensive summary of our work (see report deliverables)

4 Abstract

BirdGang is an open world, co-operative multiplayer game. As the new chiffchaffs on the block, the players must earn the Gang Boss' respect by together cleaning the streets of nuisance children and dull politicians. The aim is to ultimately save several hapless innocents from an untimely demise involving a hot-air balloon and a long drop.

The game is set on the small sub-tropical island of Gran Canaria. The players are free to explore the vibrant city comprising of colourful buildings, beautiful parks and exotic beaches. The island is inhabited by a population of 200 networked AI agents, of varying moral statures, who go about their business in the city. We use Unity's Navigation system for pathfinding, and have developed our own goal and spawn systems for agent behaviour. Between rounds, players can gain extra points by hitting bad people, who are distinguishable with a red circle. Precision is key, as near misses will cause people to panic and flee the scene. Hitting innocents with green circles will soil the Gang's good name, and is penalised to discourage careless discharge. A score is shown to keep players aware of the team's standing.

Players must remember to keep flying otherwise the flocks of other gangs on the block will encircle them. The flocks are controlled by a Boids algorithm which uses six steering behaviours: alignment, separation, cohesion, boundary avoidance, goal following and obstacle avoidance.

Given the size of the world and the number of agents, performance concerns are inevitable. We have used profiling data to optimise network and rendering performance, with custom serialisation for networked objects, and the use of custom shaders, occlusion culling, and Unity's Level of Detail system for rendering.

As a bird, the player navigates the world with our physics-based flight system, using the keyboard to accelerate and the mouse to adjust flight angle. Objectives, and the punishment of the morally bankrupt, are aided by precision defecation, the product of which soars through the air in a physically accurate manner. These controls are intuitive for inexperienced players, but also allow a degree of skill to be gained.

Players can communicate with their teammates by placing waypoints rendered as glowing beams that indicate the location of the calling player. At the same time, text is displayed on every player's screen calling their attention to the request for help. Each player can see an icon above their teammates' heads allowing them to know where their allies are at all times. To clearly present who's waypoint is who's, a colour coding system is used, where a player will have a waypoint, help text, and icon of the same colour.

The music and the sound effects make the overall experience more exciting as the rounds progress. The updates from the Gang Boss after each round informs the player how successful they are in their journey to join the gang.

The game is divided into four rounds, each starting with a cinematic cutscene to deliver story and the goal for that round, narrated by the Gang Boss. The advantage to this structure is that rounds can be added, removed, or modified without effecting the rest of the game, and allow for variety in objectives and gameplay. Our rounds are:

1. Tutorial. A guided introduction to the game's world and controls. The player undergoes basic flight training, guided by the Gang Boss through the hoops in the sky. Upon reaching an island the players encounter their first miniboss - the Naughty Child.

2. Robber. The Bank has been burgled and the alarm is going off. Players must follow the money trail to locate and subdue the elusive Robber. We implemented the money trail as a particle system.

3. Mayor. Crowds have gathered in the main square for the Mayor's speech before they all head to the carnival. However, they're just being sent to sleep with his rambling classical metaphors and alliterative slogans. Teach him that he doesn't run this city, while avoiding the deathly boring speech cloud. Again, we use a particle system for the speech.

4. Carnival. As the day draws on, the carnival arrives. But no-one told the hot air balloon operators about the gale force winds. Citizens hang from the balloons: Save them. For this mission, we implemented a Verlet rope simulation connecting the people to the balloons.

As the shadows lengthen on the city, the Gang Boss reflects on your progress: have you done enough to be admitted into Bird Gang?

5 The Team Process and Project Planning

Given the scope of the project, a successful outcome required detailed planning and a common vision. At the start, we spent time discussing our initial ideas and blueprinting how we wanted to communicate and establish an overarching plan. This step was crucial as it ensured we maintained consistent progress and met each deadline to a high standard. It also made certain all team members felt heard, respected, and free to voice their opinions without judgement.

5.1 Group development

Our project can be broken down into the four stages of Tuckerman's model: forming, storming, norming and performing [1].

5.1.1 Forming

In the forming stage of the project, we were able to establish some of the core practices that we would maintain throughout. For example, using the Scrum model (see Section 5.2) and creating a Continuous Integration and Deployment pipeline. A focus early on was learning Unity and tools and techniques we would need to develop our game. We also began to form professional relationships and gained an understanding of the group dynamics.

5.1.2 Storming

Inevitably, difficulties arose as the game was starting to take shape and visions were not entirely aligned. In order to tackle these conflicts respectfully we found it valuable to establish a culture of constructive criticism and candid feedback.

To do this, we followed the Netflix 4A's feedback guide.[2] When providing feedback, we always *aimed* to assist and give points that were specifically *actionable*, and when receiving constructive criticism, we tried to *appreciate* input without becoming defensive. We then made a conscience decision whether to *accept* or disregard feedback. We applied this when we discussed working hours. With some of our team being early birds whilst others are night owls, we initially found it challenging to find meeting times that suited all. To address this, we created a shared calendar on Microsoft Teams, populated with each person's commitments. We used this as a tool to find meeting times that could work well for everyone. We then scheduled in our twice-weekly meetings - Sprint Review and Sprint Retrospective - for the remainder of the project.

5.1.3 Norming and Performing

During these stages, we formed our core vision for the project and understood the tasks necessary to meet our goals. We used two key techniques to identify potential new features and tackle user feedback points. These were debates and discussions. Where this was not suitable, we produced rapid prototypes of multiple viable solutions and made a more informed decision. An example of this was story boarding our game and it's various rounds. We identified pros and cons of each round and were able to come to a resolution and final decision.

In contrast, reaching an agreement was not possible when discussing the world background. Its visual and subjective nature made it difficult to decide on without seeing a prototype. Four potential solutions were implemented to an Minimum Viable Product standard, making it simple to reach an unanimous conclusion.

5.2 Planning and time management

We followed the Scrum model of Agile development throughout the project [3]. We completed weekly Sprints, producing a new production build on the Friday of every week. We chose to only deploy weekly because of anticipated long build times that would make more frequent integration infeasible. We used our Monday Sprint Review to analyse user feedback, usually collected on the weekend, as well as to incorporate our weekly meeting with Tilo. We then used this to generate Sprint goals and allocate work for the week ahead. During our Friday Sprint Retrospective, we discussed what went well in the Sprint, as well as potential improvements going forward. To visualise our work and plan our tasks effectively, we made use of project management tools – specifically a Kanban Board and Gantt Chart. At the end of each Sprint Review session, we added our Sprint goals to the Kanban Board and updated them accordingly as the week progressed. This proved incredibly useful in maximising our efficiency and flow as a group.

We had planned to use the Gantt chart to map out our task deadlines and envisage critical paths. However, we soon found it did not give us the full flexibility we desired. We instead chose to use a Figma Board. Built off the same post-it note model as a Gantt chart, Figma allowed us to still visualise our critical paths and assign task priority weightings, but with added flexibility. Gantt charts had made it difficult to easily work in parallel with one another and add side tasks flexibly, whereas Figma did not follow such a restrictive framework. This left us free to add notes and connections, as well as more efficiently record task progression. We continued to use a Figma board for the remainder of the project, diagramming game plans and discussions, as well as keeping track of bugs.

Both the Kanban and Figma boards gave us a clear

and united vision of both long-term route maps, as well as short-term tasks. As a result, we were able to prevent a last-minute crunch and keep track of all feature development.

With regular meetings and communication central to the success of our project, recording the content of our discussions was vital. To do this, we wrote a detailed agenda prior to each meeting and maintained thorough meeting notes about all dialogue and actionable points. This ensured feedback and discussion outcomes were always accessible and key information was not lost in the broadness of the topics conversed. We also produced written accounts of panel feedback, to ensure ideas and potential changes were taken onboard.

When allocating tasks to individuals in the group, we took a vertical slicing approach. As we were all new to game development, we thought it best to each garner a wide range of skill sets, giving us the flexibility to change paths depending on the varying demands at each phase of the project. This also made our team more robust as we were more tolerant to temporary absences (e.g., with four members getting Coronavirus over the course of the project). We split the game into key aspects, such as flight, targeting, environment and AI, and formed subgroups within each. These subgroups were flexible and continued to shift as requirements changed. Eventually, we were able to find our personal interests and begin specialising as experts by topic, whilst still maintaining the vertical slice approach. For example, when working on front-end and modelling versus back-end and networking.

5.3 Collaborative Development

In our GitHub integration sessions we merged each team members weekly progress, ensuring that everyone had the same unified playable code to build off for the next week. This method mitigated large disparities which arose over the course of the project, as each team member was continuously working on different or occasionally conflicting segments of the code.

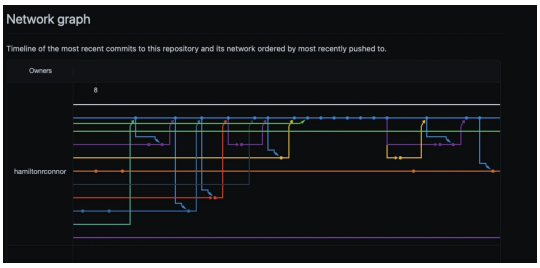


Figure 1: Image showing network graph of GitHub branches

As a team we made over 1300 commits to our GitHub. This repository comprised of: main, devMain, production and several temporary feature-based branches. During each integration session, when merging into devMain, we performed regular code reviews to ensure that new code met our expected standards. The devMain branch held the collective code changes done by all members over the week and served as the proxy for the main branch where we could perform integration, performance and stress tests (see Section 7.4. This maintained the smooth functioning of the game at each iteration. Once these tests passed, we merged changes onto our main branch. This triggers the start of our Continuous Integration and Development pipeline (see Section 7.3) which, if tests pass and the build is successful, deploys our code into our production branch that runs using GitHub pages.

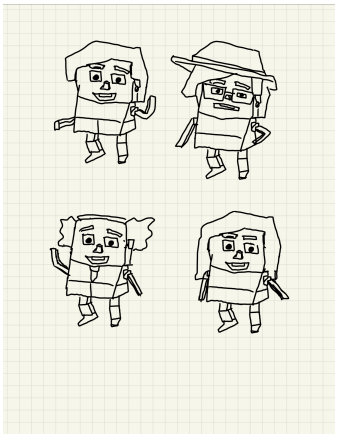


Figure 2: Initial designs

To ensure collaborative development when creating the assets for the game, we regularly conducted team feedback sessions on initial sketches and model ideas (Figure 2). This allowed all members to voice their opinions, and prevented unnecessary development on assets that would not be taken forward.

Furthermore, a core approach we used was to practice paired programming (two members of a team develop code working on the same machine at the same time) which is an agile software development technique. This provided instantaneous code reviews and allowed members in a subgroup to discuss ideas. This helped us to build a strategic direction of work, raised the communication standard within the project and develop professional relationships within the group.

5.4 User Feedback

When developing our project, we took a User-Centred Design approach (Figure 3). This kept the needs of potential end users at the forefront throughout, helping us gather key insights and make decisions not based solely on bias views from us as developers. To collect this continuous feedback, we created a comprehensive Google Form that we then asked peers to

complete, after playing a full run-through of our game. This feedback was collected throughout different stages of our development process and helped us steer our discussions and feature decisions accordingly.

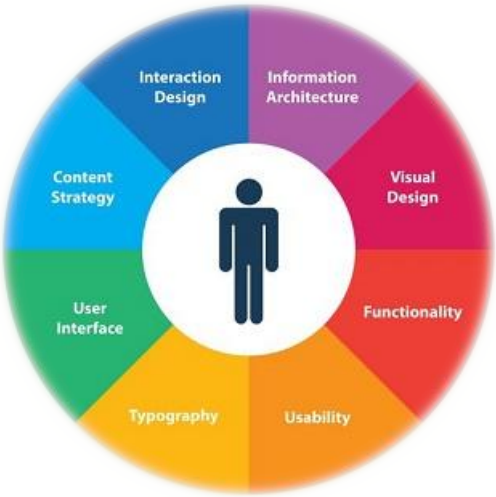


Figure 3: User Centered Design approach

A key example of this was when deciding on our flight mechanics. We asked peers whether they preferred controlling the flight of the bird using mouse movements, keypresses, or both. The answer weighted heavily in favour of both, with a slight preference for mouse over keys. This differed to what we had initially expected to implement as a group with just key controls, and so it was invaluable to have the feedback from a wider group of people. We listened to the feedback and made both control schemes available, as outlined in Section 8.2). Following on from this we performed another round of user feedback and were pleased with the results as they showed a high level of satisfaction with the new combined control scheme.

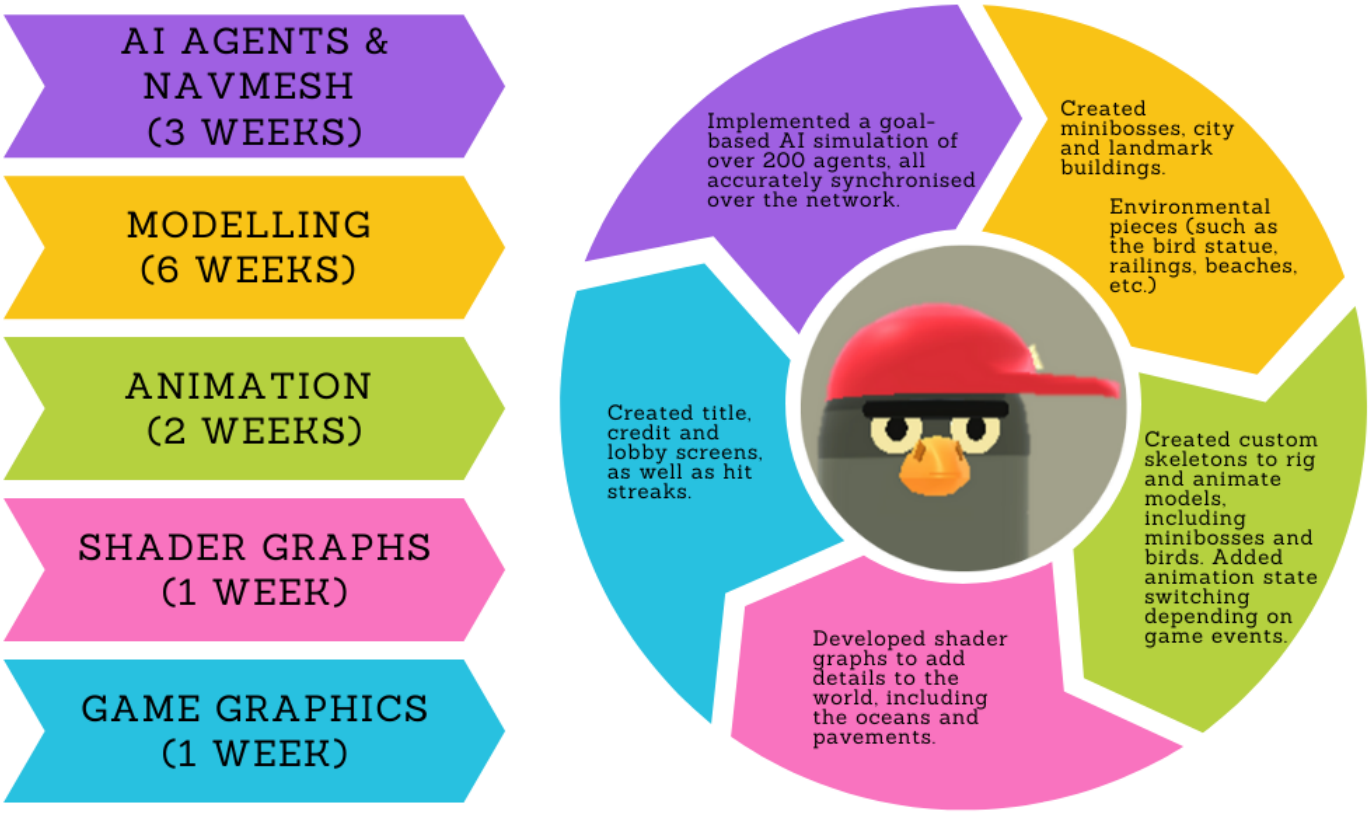
We also placed a strong emphasis on panel feedback at each formative review stage. As the user group of panel members varied significantly from that of our peer reviewers, it provided us with a different and often conflicting perspective on many aspects of our game, particularly regarding its difficulty and ease of use. We took all feedback seriously and implemented changes where appropriate. A notable example of this was with regards to our initial plans to implement a mini-map feature that could help locate other players in the world. This was something the review sessions indicated was not favourable among panel members. They instead wanted a feature that tied in more closely to our existing game mechanics. Therefore, we went back and developed our glowing waypoint system and colourful player icons (see Section 8.10). We also included establishing shots of our world when panning during Cinemachine cutscenes at the start of each round. These feature changes proved successful and a more appropriate solution with the users in mind.

5.5 Final Reflections

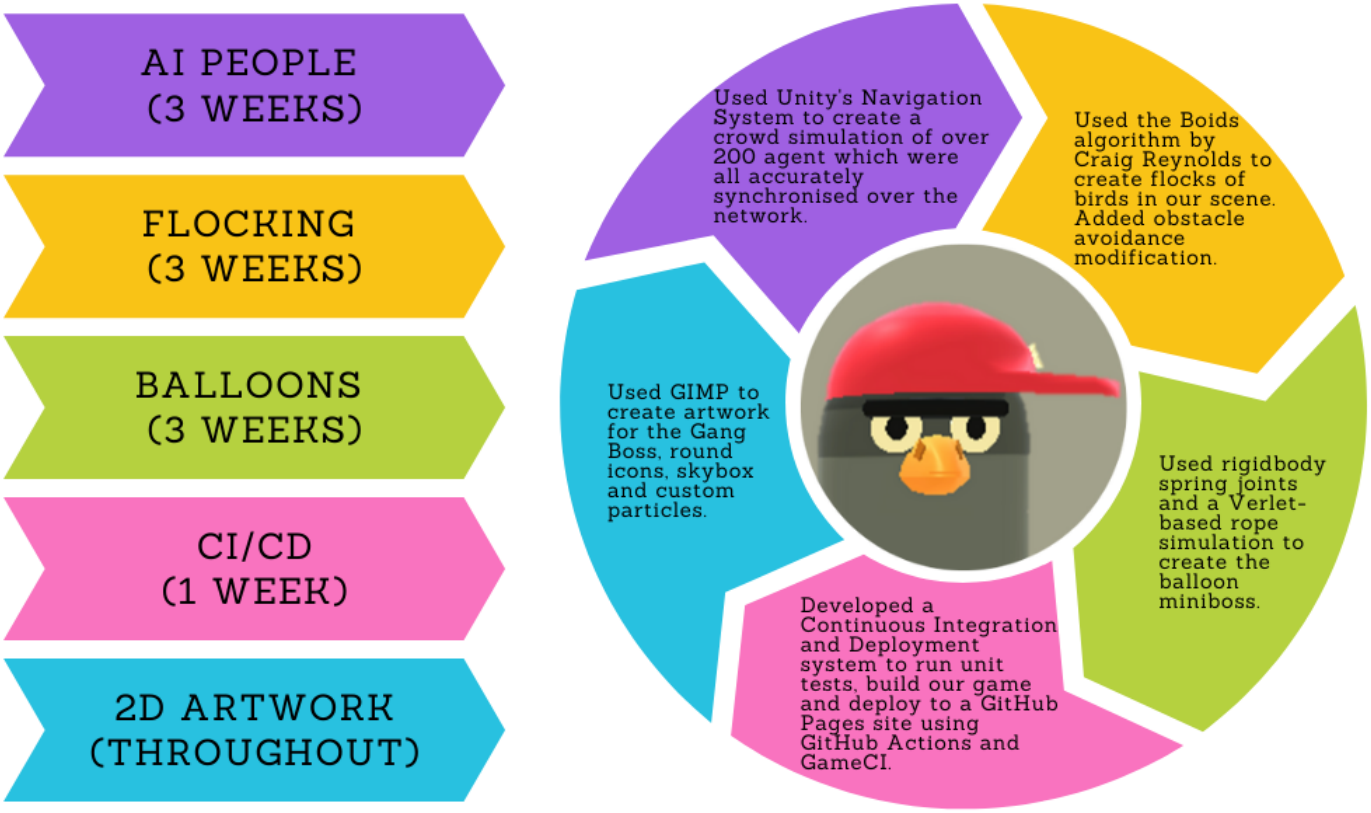
Overall, we learnt that agile development tools must be strategically deployed to be most effective. Misdirected efforts simply hinder the progression of the project rather than enhance it. Furthermore, remembering to maintain focus on the target audience is of most importance. Although opinions will vary depending on the reviewer, narrowing focus to the intended goals is key. We also learnt that in team projects, collaboration is vital to produce a cohesive product and maximise a group's potential. Our final takeaway is that in large projects some level of conflict is inevitable; the success of a project is measured on resolving these conflicts effectively, not attempting to avoid them altogether.

6 Individual Contributions

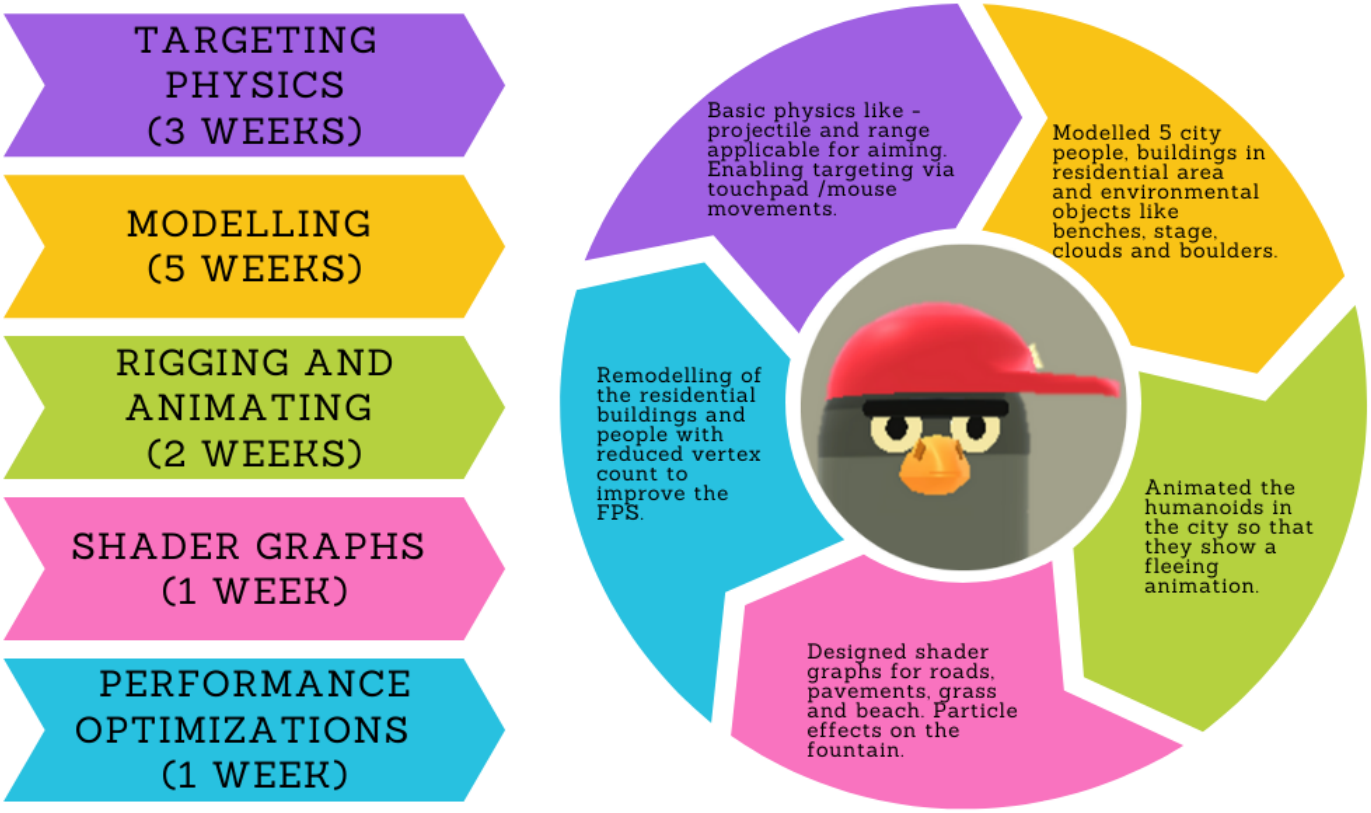
6.1 Jaya



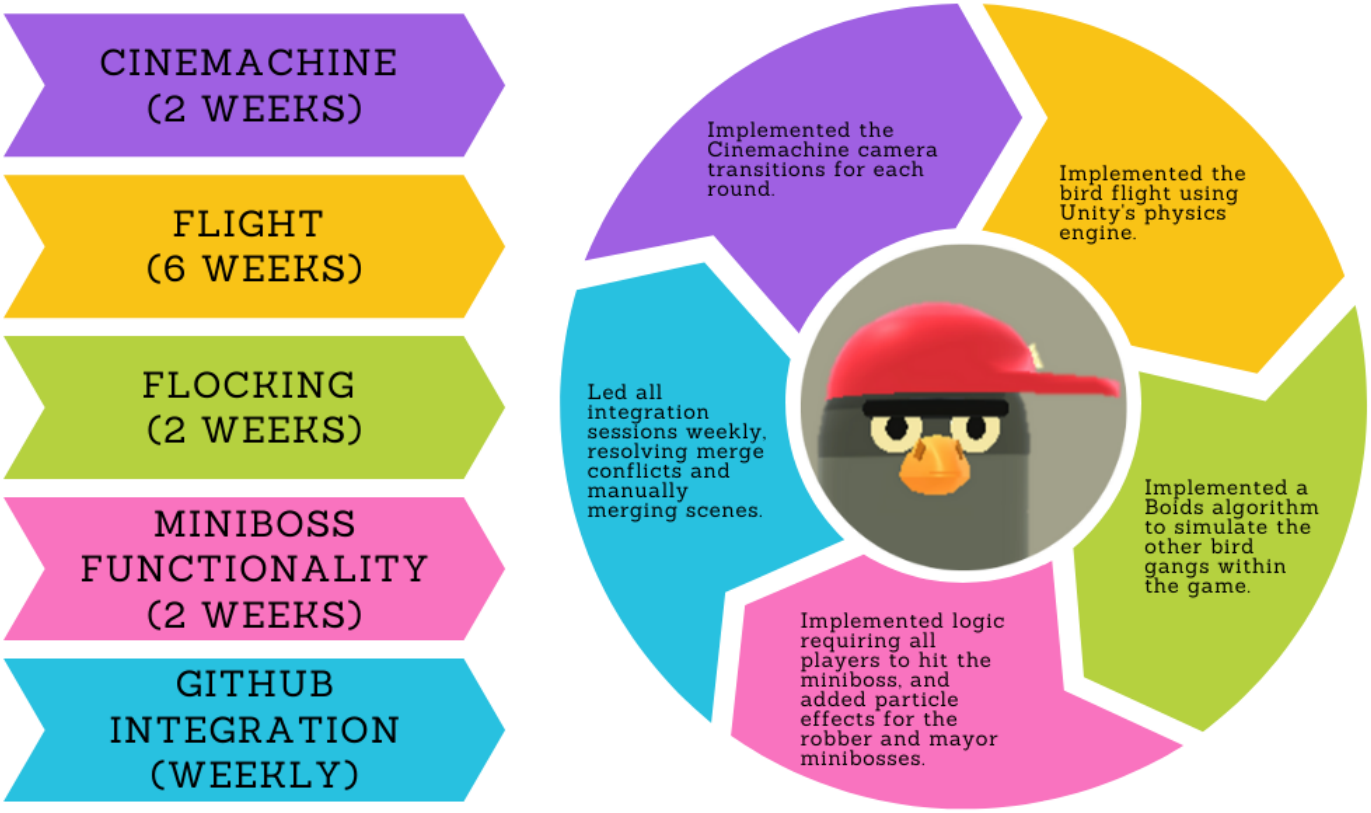
6.2 Connor


AI PEOPLE
(3 WEEKS)
FLOCKING
(3 WEEKS)
BALLOONS
(3 WEEKS)
CI/CD
(1 WEEK)
2D ARTWORK
(THROUGHOUT)
Used Unity's Navigation System to create a crowd simulation of over 200 agent which were all accurately synchronised over the network.
Used the Boids algorithm by Craig Reynolds to create flocks of birds in our scene. Added obstacle avoidance modification.
Used rigidbody spring joints and a Verlet-based rope simulation to create the balloon miniboss.
Developed a Continuous Integration and Deployment system to run unit tests, build our game and deploy to a GitHub Pages site using GitHub Actions and GameCI.
Used GIMP to create artwork for the Gang Boss, round icons, skybox and custom particles.

6.3 Tanvi


TARGETING PHYSICS (3 WEEKS)
MODELLING (5 WEEKS)
RIGGING AND ANIMATING (2 WEEKS)
SHADER GRAPHS (1 WEEK)
PERFORMANCE OPTIMIZATIONS (1 WEEK)
Basic physics like - projectile and range applicable for aiming. Enabling targeting via touchpad /mouse movements.
Modelled 5 city people, buildings in residential area and environmental objects like benches, stage, clouds and boulders.
Animated the humanoids in the city so that they show a fleeing animation.
Designed shader graphs for roads, pavements, grass and beach. Particle effects on the fountain.
Remodelling of the residential buildings and people with reduced vertex count to improve the FPS.

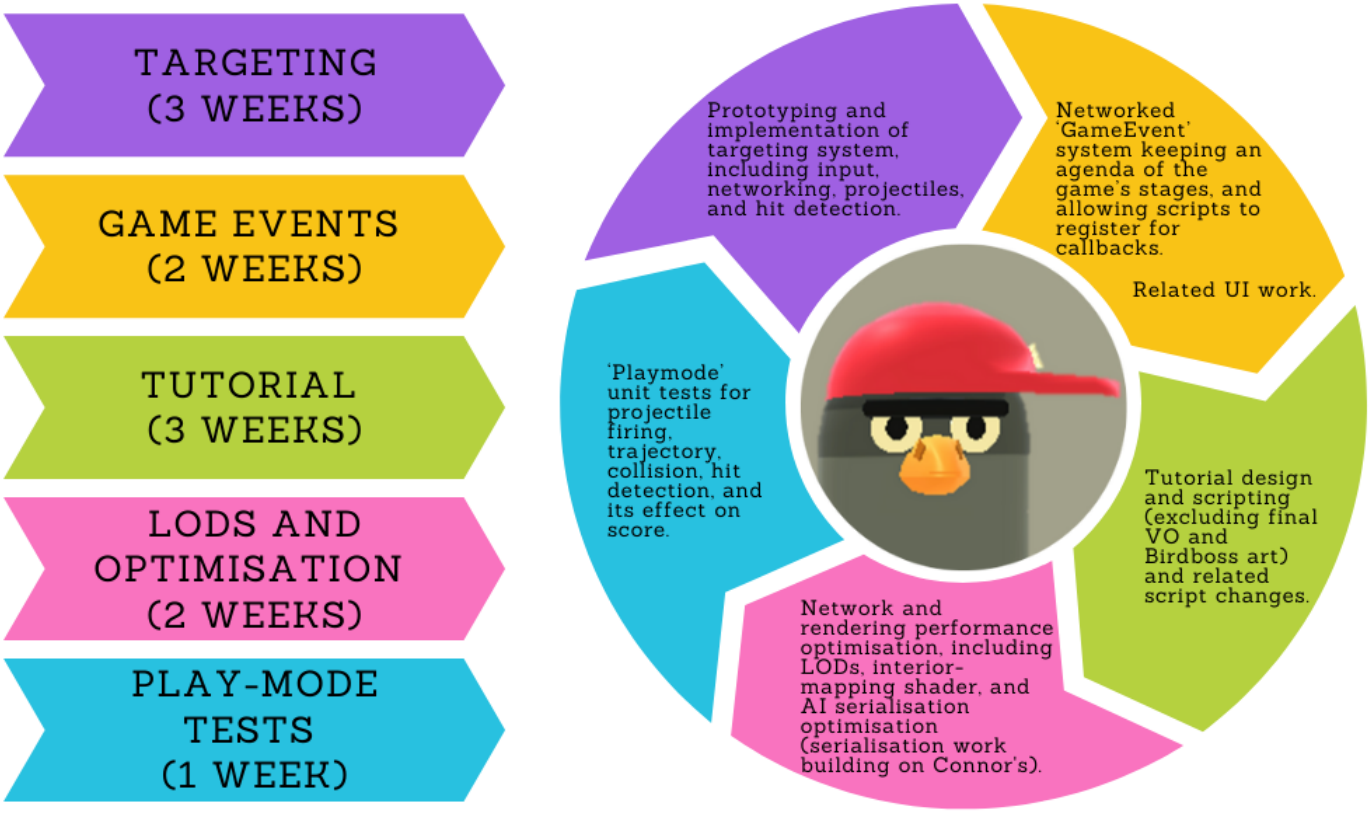
6.4 Ambika


CINEMACHINE
(2 WEEKS)
FLIGHT
(6 WEEKS)
FLOCKING
(2 WEEKS)
MINIBOSS
FUNCTIONALITY
(2 WEEKS)
GITHUB
INTEGRATION
(WEEKLY)
Implemented the Cinemachine camera transitions for each round.
Implemented the bird flight using Unity's physics engine.
Implemented a Boids algorithm to simulate the other bird gangs within the game.
Implemented logic requiring all players to hit the miniboss, and added particle effects for the robber and mayor minibosses.
Led all integration sessions weekly, resolving merge conflicts and manually merging scenes.

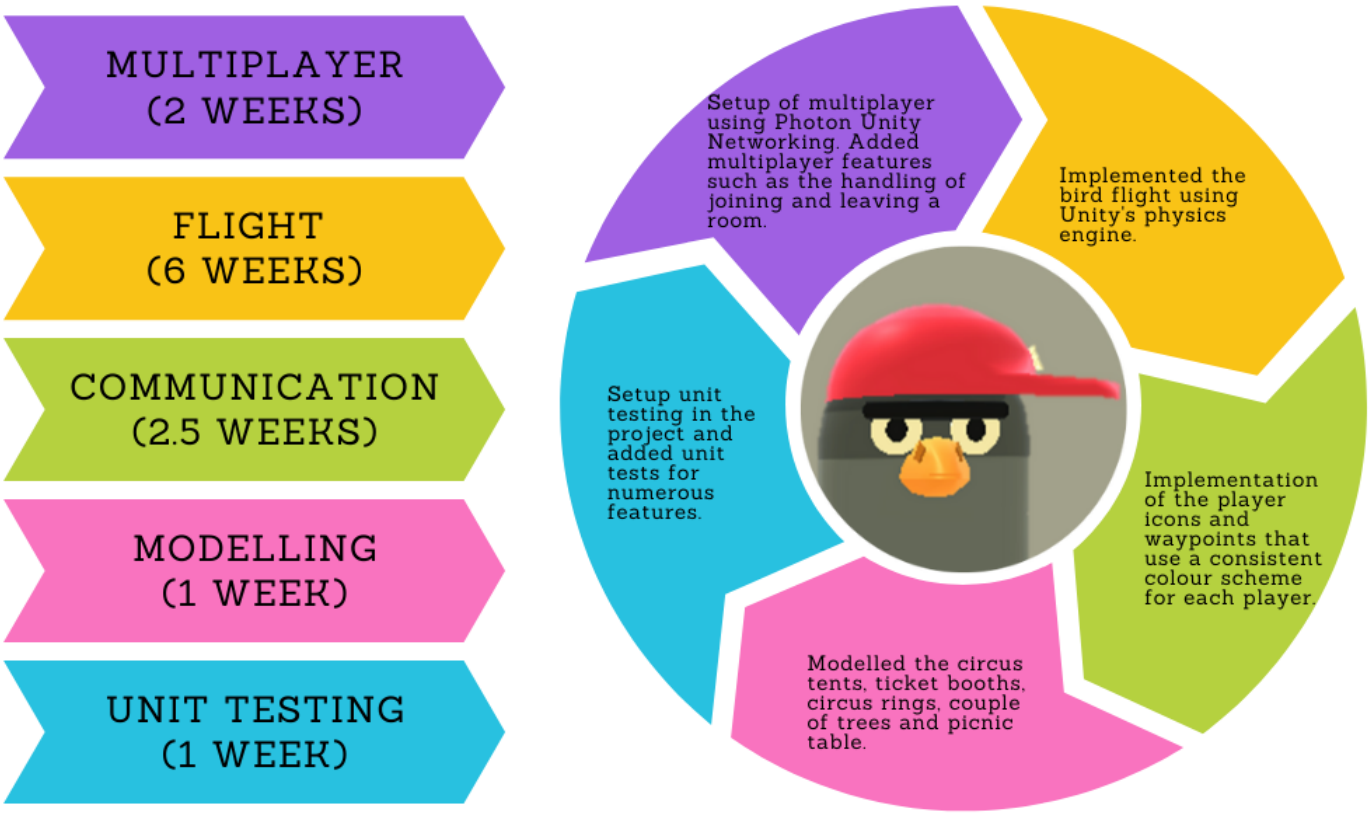
6.5 Iban


FLIGHT
(6 WEEKS)
PHYSICS
(2 WEEKS)
MINIBOSS
BEHAVIOURS
(1 WEEK)
INITIAL LIGHTING
(1 WEEK)
AUDIO
(2 WEEKS)
Implemented the bird flight using Unity's physics engine.
Implemented various physics aspects to the bird such as: hovering, wind, building collision.
Initial coding for the mayor speech particles pushing player back and robber disappearing.
Set up initial lighting for the storm stage.
Implemented dynamic sound effects and outcome-based voiceovers.

6.6 Joe


TARGETING
(3 WEEKS)
GAME EVENTS
(2 WEEKS)
TUTORIAL
(3 WEEKS)
LODS AND
OPTIMISATION
(2 WEEKS)
PLAY-MODE
TESTS
(1 WEEK)
Prototyping and implementation of targeting system, including input, networking, projectiles, and hit detection.
Networked 'GameEvent' system keeping an agenda of the game's stages, and allowing scripts to register for callbacks.
Related UI work.
Tutorial design and scripting (excluding final VO and Birdboss art) and related script changes.
Network and rendering performance optimisation, including LODs, interior-mapping shader, and AI serialisation optimisation (serialisation work building on Connor's).
'Playmode' unit tests for projectile firing, trajectory, collision, hit detection, and its effect on score.

6.7 Emily


MULTIPLAYER
(2 WEEKS)
FLIGHT
(6 WEEKS)
COMMUNICATION
(2.5 WEEKS)
MODELLING
(1 WEEK)
UNIT TESTING
(1 WEEK)
Setup of multiplayer using Photon Unity Networking. Added multiplayer features such as the handling of joining and leaving a room.
Implemented the bird flight using Unity's physics engine.
Implementation of the player icons and waypoints that use a consistent colour scheme for each player.
Modelled the circus tents, ticket booths, circus rings, couple of trees and picnic table.
Setup unit testing in the project and added unit tests for numerous features.

7 Software, Tools, and Development

7.1 Software developed

7.1.1 Unity

A key consideration in choosing what game engine to develop in was the ease in which we could implement our flagship technologies. This meant searching for a game engine that supported both physics and AI. The game engines we considered using were Unity, Unreal Engine and Godot. Initial research into Godot showed that it had limited functionality to make a 3D game, thus this game engine was also ruled out [4].

This left us to choose between Unity and Unreal, both game engines that are widely used [5]. As they both use the same physics engine, PhysX, either would have been a suitable choice. Given the timeframes and depth of this project, we wanted to use a game engine that was intuitive to use and simple to pick up as we developed. Our research showed that Unity is more suited for beginners. A notable advantage of the Unreal Engine, however, is the graphics rendering and the ability to produce hyper realistic games. Given the style choices of our game we felt we could produce the same quality graphics in Unity, since hyperrealism was not a style we were looking for. Instead, we wanted to produce a low-poly, vibrant and cartoon looking game. Having considered the advantages and disadvantages of both game engines, we chose to use Unity.

Throughout the development process we used a number of Unity features, such as the Unity Profiler, the Unity Navigation System and the Unity Particle Effects System. However, there were some limits to the features that Unity provided. The most notable of which was its interaction with GitHub when merging scenes. Due to the setup of Unity's scene files, GitHub was unable to merge scenes. Therefore, we created a new scene for each new feature we were adding, and manually merged these. Another limitation we faced was the stability of the Unity Editor, despite using a long-term support version. While no work was lost when the Editor crashed, it did disrupt the workflow, resulting in periods of slowed down development.

7.1.2 Photon

There were several key factors that led us to use Photon as our networking solution. Firstly, we knew from the outset that a room-based multiplayer system would be ideal for the type of game we were producing. Secondly, Photon abstracts away communication with the server to a level which allows us to spend our time adding features, rather than fixing problems with network code. Thirdly, Photon is highly scalable, whilst we are only able to support a maximum of 20 concurrent users on the free tier, Photon has the capability to scale to tens of thousands of users. Finally, in comparison to Mirror Networking (an alternative option to Photon) and Unity's own networking solution, Photon has the best documentation and is the easiest to learn.

The specific version of Photon we chose to use was Photon Unity Networking (PUN) 2. The justification for this decision was that while Photon Fusion had some additional features, it is less supported due to its recent release in 2021. Therefore, we felt we would struggle to reach the point where we would be able make best use of the features.

Our networking system used a Master Client solution, where the player that creates the room has definitive authority over the game and its state. So, the master is responsible for starting the game, performing the simulation of our AI agents, and spawning and despawning networked elements of rounds. To minimise network traffic, whatever processing can be performed on the client, is, and only minimum data is serialised.

For example, round progress is entirely deterministic, down to the time the game begins. Therefore, cutscenes and spawning and despawning of non-networked per-round elements can be handled on the client without any network intervention. The same goes for agents, where their next goal location can be serialised when it changes, and pathfinding can then be handled client side.

Of course, in some cases this leads to divergence between the master and clients. We allow this only if it is acceptable from a gameplay standpoint, and after ensuring that these changes cannot lead to knock-on effects to other networked GameObjects. For instance, bird poo's position may diverge completely between clients, but hit detection and score changes are always taken from the client who fired it.

7.2 Software development and maintenance

During development we maintained our software using GitHub. Figure 4 shows our weekly development process. We chose to integrate on a weekly basis due to the substantial amount of development that would take place between seven people.

At the beginning of the week, all developers would pull from the main branch and create a new branch on GitHub based on main. During the week, developers would add new features and models, and edit existing features as required. At the end of the week, all branches were integrated on GitHub. We had a separate development branch (devMain) into which all the changes were merged. This branch was used as we did not wish to deploy to WebGL with every merge we did. Due to the nature of the project, merge conflicts also

occurred often. We fixed these using GitHub's resolve conflicts tool, which identified the scripts that were conflicting and the changes that had been made. Once all branches were merged into devMain, we manually merged scene changes. Our use of prefabs, as outlined in Section 8.8.6, made merging scenes simpler as each prefab only needed to be added to the new scene, and no configurations made. Following this, obsolete scenes, scripts and prefabs were deleted from the Unity project, along with old branches on GitHub to minimise risk of confusion and errors. In-house testing was then conducted. Minor bugs were resolved in the meeting and major bugs assigned to a developer for the following week. Once the team were happy with the state the game was in, the devMain branch was pushed to Main, triggering a deployment to WebGL.

7.3 Continuous Integration and Deployment

From the outset of the project, we understood the importance of having a Continuous Integration and Deployment pipeline. Figure 5 shows a diagram of the pipeline we implemented in week two of the project. There were several key advantages to working in this manner. Firstly, by running tests before deploying we were able to ensure that only reliable code made it to production. Secondly, by using GitHub Pages and Continuous Deployment we were able to establish a near zero downtime deployment to a WebGL build, meaning that we could gather user feedback as often as possible. This also meant we could see the product that we were developing, which helped with group motivation.

To build our game we used GameCI Unity builder because it works well with GitHub Actions and has a strong community and documentation. GitHub Actions is comparable to other pipelines like Google Cloud and AWS in terms of capability. However, by using GitHub Actions we were able to take advantage of the full GitHub ecosystem for ease of deploying to GitHub Pages.

7.4 Testing Strategies

Across the project we deployed several testing strategies allowing us to minimise the chance of bugs in our game.

7.4.1 Unit Testing

The integration of seven people's worth of work meant that it was important for us to have a system of unit tests in place to ensure that new code additions were not breaking existing functionality. Unity offers a testing framework using the NUnit library and offers two different methods for testing our code - editor tests and play tests [6]. Where tests did not involve user input, we used editor tests to save build time and size [7]. For tests involving user input, we had to use play tests.

We split up the tests into separate files that were named according to the script they were testing. This scheme ensured that any issues that arose resulting in failed tests were a quick process to identify. It also meant that as new features were added, or new bugs were discovered, it was a simple task to add more unit tests.

7.4.2 Integration Testing

Testing of this nature was performed towards the end of our integration session, before we merged to devMain. These tests involved verifying that new features worked with existing code and especially with the other features added that week. We would identify a set of expected behaviours from the new features and by playing through the game or specifics parts of the game assert that these were met. Of all the types of testing we performed, this uncovered the most bugs and errors.

7.4.3 Performance Testing

Throughout the project we maintained an awareness of the current performance of our system. By doing this we were able to catch mistakes from going into production, for example, at one stage the culling mode of our AI people was incorrectly set, resulting in a drop in frame rate due to animation updates being performed on occluded meshes. One area we maintained a particular focus on was the performance of our networking system. To stress test this, we used the Photon Lag Simulation GUI component to add lag, jitter and message loss, and monitored the performance of various components, such as the position of our AI agents.

7.4.4 Production Testing

We completed production testing as a final stage of our integration meetings. This allowed us to identify bugs not visible in the Unity Editor. An example of this was when a `UnityEditor` method was used instead of an equivalent `UnityEngine` method. This resulted in missing functionality that was not picked up and made it into production. By doing this style of testing we were able to resolve bugs before end-users were exposed to them.

7.5 Game Extensibility

The round-based structure of our game allows for new functionality to be added and for our game to easily be extended.

The four rounds, as well as the introduction and finale are all managed by an events system. For a new round to be introduced, it can be added to the `Stages enum`, and then to the game agenda along with its length. Each round also has a manager, in which all the changes for that round occur. For example in our

`RobberManager` script, a robber is instantiated, and the bank alarm is turned on. In addition to adding the new round to the agenda, a new manager can be created, which can register for callbacks from the event system.

Each round is introduced by a cutscene. To add a new camera transition to and from the cutscene, a new Cinemachine Virtual Camera can be added as a child of the State-Driven Camera, and a new state can be created in the Animator controller. The change of state can then be triggered within a script. The time, the path and the style that the camera takes to get from the previous camera to the new camera can be defined in the Cinemachine Blender Settings.

To extend our game further, the island can be easily modified. Our use of prefabs allows configured GameObjects to efficiently be added into the scene and positioned. Next, spawn points can be added to our custom spawning system. As all agents follow a NavMesh, this can be re-baked to fit a different island. We used the NavMesh Surface component which allows linked NavMeshs. This would allow future developments like a ferry system or additional islands.

We have already added some functionality to extend our game beyond the standard 10 minutes. When the player reaches the credits screen, a button will appear for the master client, allowing them to continue the game. This will take all players back into the game where there will be no more rounds, but players are free to try to get their score as high as possible, or practice their flight skills with their teammates. As the spawners keep the numbers of good and bad agents constant, the game will only end when any player leaves. Alternatively, the master client can choose to go back to the main menu where they can start the game from the beginning.

Throughout our project we have ensured our code is commented, and that variable and script names are intuitive, and all follow the same convention. We have also split up scripts wherever possible, for clarity and to enable us to add tests. As a team, we have often needed to build upon our teammates' code, thus we are confident that all code is clear and understandable to someone with coding knowledge. Furthermore, we have consistently chosen tools that are widely supported, such as Photon PUN.

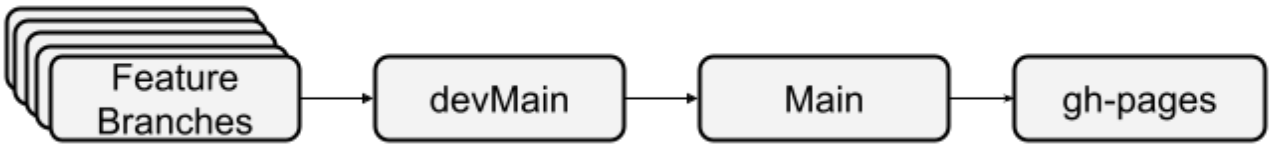


Figure 4: Weekly development process

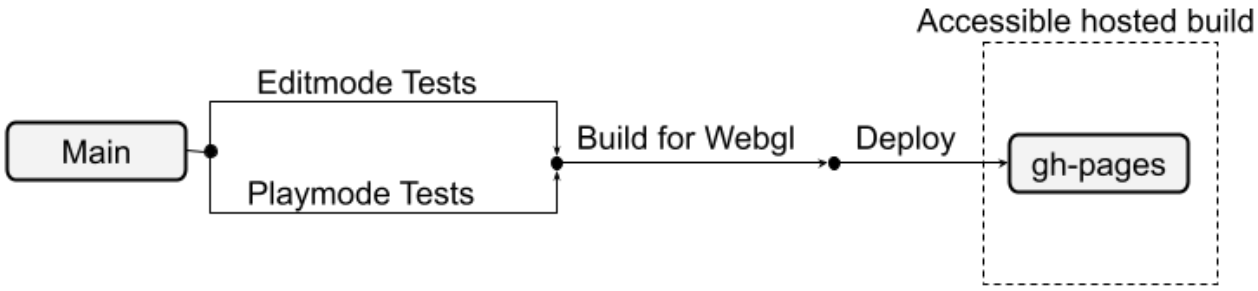


Figure 5: Continuous Integration and Deployment pipeline

8 Technical Content

8.1 Game Events

Given the game is centred around a series of events, keeping scripts in time with the current stage is essential. To avoid a disparate collection of timers and dependencies, we implemented a central system to keep time, and inform scripts when stages start, end, and progress.

A script may register for its selection of callback types and stages from a singleton. These callbacks allow for per-event spawning and de-spawning of assets, keeping UI elements aware of game progress, and other triggered events, such as cutscenes and music.

Keeping these timings consistent across the clients is essential, as differences would lead to inconsistent game state. This is especially important in a social game. Firing Remote Procedure Calls (RPCs) when stages begin involves significant latency, and so is not suitable. Instead, the master client keeps an agenda of each stage and its respective length, which it sends to clients at the start of the game. When every client has loaded, it also sends an epoch network timestamp, which all time measurements are made relative to. Using network timestamps rather than a variable offset from local time prevents client-to-client drift, something we have observed with Unity's own timekeeping. After this initial setup, timekeeping and callback firing is entirely clientside, with no latency or drift.

This system keeps each player's game in sync, keeps code complexity down, in turn making code maintenance easier, and allows stages to be trivially added, reordered, and removed for development and testing.

8.2 Flight

8.2.1 Process of developing our control scheme

Given how the user experience is centred strongly around flight it was important to prioritise this flagship feature, thus we dedicated three group members to its development (flight team). Having three group members of varying gaming experience working together to create the control scheme allowed us to formulate a system that drew from everyone's viewpoints and experiences.

Each week the flight team presented a prototype of the control scheme to the rest of the group. This was a good opportunity to get fresh feedback and allow all group members to have their own influence on the feature.

The first prototype control scheme that we then presented to panel members solely used the keyboard. Whilst our feedback was positive overall, we observed that the reviewers only used one keypress at a time. This resulted in character movement that was not representative of the fluid movement we see in real birds. Hence, we sought to extend our control system to utilise the mouse/trackpad.

The second prototype control scheme incorporated this extension, where the mouse/trackpad was now used to steer the character and the keypresses had been reduced to only use W. With this new scheme, character movement was more closely related to real bird movement. Again, we received positive feedback on the control schemes ease of use. However, it was also pointed out to us by some reviewers with gaming experience that turning around could be quite a long process which could be frustrating. They proposed the reimplementation of the A and D key to provide alternative turning options.

Our third prototype control scheme built upon our previous prototypes and encompassed the panel's feedback by extending A and D to allow for turning on the spot. The feedback we received in this round of reviews led us to implement a push back motion using the S key.

Aside from the panel's feedback, we also researched existing bird flight games and read through many reviews to assess what players liked and disliked about existing control schemes. A key trend we identified was that in the absence of dynamic speed, players often got bored flying around. Consequently, the final control we added was acceleration, activated by pressing the SPACE key. Over time this acceleration coefficient is reduced to zero, simulating drag and returning the character to baseline velocity.

8.2.2 Use of physics

Attached to the bird character is a rigidbody component which allows it to react to real-time physics, such as gravity, mass, drag and other forces. This wasn't something we could do with the CharacterController component, an alternative to a rigidbody component. We used the rigidbody to create the forward motion of the bird through the world, as well as its backwards motion. To maintain realism, it would not make sense for the bird to be able to reverse backwards in the same way it can move forwards. Nevertheless, user feedback informed us that some backwards motion was required. As a balance between playability and realism, we implemented a pushback motion. This is activated using the S keypress which uses an instant force impulse that acts on the rigidbody component.

To create a gliding motion with mouse/trackpad turns, we used Quaternion Euler angles to calculate the new rotation. This method provides smooth interpolation which results in fluid movement. It is not recommended to directly interact with Quaternions since they are based on complex numbers, therefore, to implement A and D we used Torque since for this

turning motion we only needed to calculate a rotation around the y-axis [8].

8.2.3 Environmental Effects

To implement realistic interaction with the world we utilised Unity's physics engine to add a number of environmental effects.

The first of which was gravity. In real life, birds do not just stop moving and therefore we needed to implement an effect when the player was not pressing any keys. One option we considered was letting the player slowly sink to the ground. However, after some testing we decided this did not provide a good gameplay experience. Players were frequently having to look up at the sky to gain height, rather than being able to look down and shoot the agents. The second option we tried was hovering. With this we take into account the bird character's mass and gravity to calculate and apply a periodic upwards and downwards force, keeping the player at the same height. Following positive user testing for this option we decided to keep it.

The next interaction we added was wind. This is used to the bring the storm in the carnival round to life and add difficulty to the final round. If the player is not moving for three seconds, they feel this environmental effect. For this we use a Constant Force component, which adds a relative force of 30 Newtons. To simulate the real-world, the direction in which this force is added is random.

The final interaction we added is seen when the player collides with objects in the world, such as a building. During our user testing, we observed that the panel struggled to recover after crashing into an object. To overcome this, we added a push back motion, which works in the same way as the S keypress. Pushing the player away from the object provides them with enough space to alter their flight path around the object. We used Unity's tagging system to define which objects in the world caused this push back motion. We chose not to add it to all objects to avoid the feature becoming a nuisance.

8.3 Targeting and hit detection

Like our flight model, our targeting uses Unity's physics engine. This simplifies firing and collision of projectiles.

For our targeting, we trace a ray from the camera's position towards the cursor in world-space, until it hits world geometry. We calculate projectile time-to-hit as the distance from our bird to this point, divided by some constant (which is an average velocity).

On the horizontal axes, we assign constant velocity. On the vertical axis, we assign an initial velocity, which varies linearly with the distance to the target, then solve for a constant acceleration (gravity) so that the projectile hits the target. For distant targets, we assign positive vertical velocity, so that the projectile follows a parabolic curve. For near targets, we give negative initial velocity, so that the projectile quickly hits the target.

When we have the velocities and acceleration, it is simple to render a targeting arc for the player with a Line Renderer using equations of motion, and allow firing of projectiles by network instantiation of rigidbodies with the correct values.

Projectiles are spawned across the network and independently simulated on each client. As they are fast moving, position synchronisation, even with lag compensation, would not be appropriate. This lack of network synchronisation means that projectiles may diverge across clients, but this does not affect gameplay or correctness.

Unlike most other systems in our game, hit detection is handled on the client who spawned the projectile, as opposed to the master. This is especially important as agent positions may differ somewhat between clients, and players should not feel that their shots unfairly miss. When a projectile hits an object marked 'target-able', it calls an `OnHit` method on that object, which can either be an RPC to the master, or a method called locally.

Our projectile system uses two layers of collider: world colliders and target colliders. World colliders interact with buildings and the ground whereas target colliders interact with people and minibosses.The target collider is larger than the world collider. This gives a small amount of lenience to the player when trying to hit a target, while still allowing the player to shoot round tight corners.

8.4 Balloon miniboss

From a physics perspective the balloon has two main components: the spring and the rope.

By using a rigidbody spring joint attached to the top of a high mass person and base of a low mass balloon, we can simulate the behaviour of a balloon tied to an anchor. This system allows for believable features, such as the balloon bouncing and spinning by when hit by poop.

In addition to this, we need to render a rope connecting the person to the balloon. The simplest solution is to use a line renderer, connecting the balloon and the person. However, this does not look believable. Ropes need to bend and deform according to gravity. To achieve this, we have created a real-time rope simulation.

This is a Verlet-based simulation [9]. The reason for using this method is that it is more stable compared to other methods, such as Euler Integration, and is accurate enough to look believable in our game without causing performance issues.

To create our simulation, we break our rope down

into N joints, where N varies based on the length of the rope. Our rope simulation has three stages:

- Simulated – Calculate the next position of the joint based upon its velocity (found by subtracting the current position from the previous position) and gravity.
- Constrained - Repeatedly apply the simulation constraints. In our simulation these are: the first joint must be on the balloon base, the last joint must be on the anchor, and we must keep a fixed distance between neighbouring joints.
- Rendered - Render the updated position of the rope on every frame.

8.5 AI People

Our city is populated with good and bad Non-Player Characters (NPCs). To bring these people to life they must walk around and interact with each other. To achieve this, we use Unity's navigation system, which moves agents towards a goal location, following a path defined by the A* algorithm [10].

Goal locations are placed around building entrances, in parks and other destinations that mirror the real world. When an agent gets near its goal, it will select a new random goal location. To create realistic movement, we have penalised movement along roads, as this gives a more natural looking feel to their movement.

To spawn our agents, we have a custom spawning system. This system allows us to easily control the number of agents of different types, bias spawn frequency around the map, and ensures flexibility and extendibility for future changes.

Before the Mayor Speech cutscene, a crowd forms around the stage. This formed a technical challenge for us, as tightly packed crowds are a feature the Unity navigation system struggles with due to a fixed maximum of eight objects considered at any one time by the obstacle avoidance system. This causes jittering, jumping and erratic movement. To avoid this, we created an area for the agents to crowd in and gave the agents varying priority levels to solve agents competing for the same spot when moving through the area. We set the mayor to have maximum priority so that he pushes other agents out of the way as he leaves the scene, in fitting with his character.

When a player poops a fleeing response is triggered in all nearby agents. This serves two purposes: firstly discouraging spam shooting, and secondly watching agents run away from poop is highly enjoyable. To achieve this, we calculate a vector in the opposite direction to the site the poop hits and then find the nearest point outside of the flee radius but still on the NavMesh that the agent can flee towards. A major challenge of this feature is ensuring accurate syncing and lag compensation across the network. Simply using a Photon Transform View results in far too much network traffic to allow believable movement, even with lag compensation. Instead, we send the next position on the NavMesh that the master agent is moving to. On clients, the agents can then locally calculate a path towards this destination. This strategy has several advantages. Firstly, we avoid sending data which is calculated locally, such as rotation. Secondly, we can avoid sending updates when the next position has not changed, rather than every network tick. Thirdly, it ensures that that the client agent is not just following behind the master agent, but is instead somewhere between the previous position and the next position, which averages to an accurate position. We also compensate for network latency between a message being sent and received, by using time delta and velocity to extrapolate agent position on an update. This allows us to accurately synchronise the position of over 200 agents in our world and has the capacity to synchronise over 800 agents.

8.6 Flocking

To continue the immersive experience in the skies as well as on the ground, we created a second type of NPC - flocking birds. If a player hovers in one spot for too long they will be targeted by a flock, which will encircle the player, distracting them and thus making it harder to aim at a target.

To create beautiful, lifelike and dynamic flocking motions, we use a modified version of Craig Reynolds', Boids algorithm [11]. The movement of each bird is defined by three steering behaviours:

- Alignment – attempts to steer towards the average direction of the neighbouring birds.
- Separation – attempts to avoid crowding.
- Cohesion – attempts to move towards the average position of the neighbouring birds.

The neighbourhood of a bird is defined by a radius, from the bird and an angle. As shown in Figure 6.

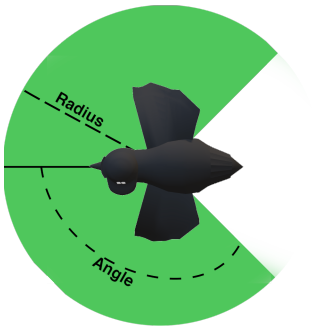


Figure 6: Diagram illustrating a bird neighbourhood

We also extended the algorithm by developing three additional steering behaviours. They are:

- Boundary avoidance – If a bird gets too close to the world borders it will attempt to move towards the centre of the map.
- Goal targeting – Each bird will attempt to steer towards a goal position. This is either the `FlockManager` which moves randomly around a set area in the map, or the stationary player that the bird is attacking.
- Obstacle avoidance - This is based on a modified version of the Curb Feeler approach proposed in [12].

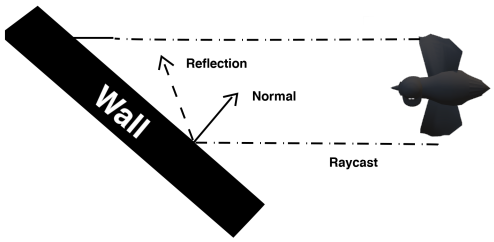


Figure 7: Diagram showing our obstacle avoidance system

The flocks in our game are of two types. Firstly, tight flocks, these birds are designed to fly in small tight groups and form the flocking patterns that we typically imagine when we first think of a flock. These birds are set to fly above the height of buildings allowing the player to better see these structures. The second type of birds are the loner birds, which still exhibit flocking behaviour but are more spread out than the other birds. They are often seen much lower than the tight birds moving in between buildings. Only the loner birds use obstacle avoidance as it is more computationally intensive to run than the other steering behaviours. We came to this decision as we felt that not having obstacle avoidance on the tight birds did not affect the game play.

Figure 7 shows how our obstacle avoidance system works. We use two `Physics Raycasts` either side of the bird to identify nearby objects. We pick the closer of the two hits and calculate the reflected angle from the object. Using this information allows us to move around the object without the birds' wings clipping into a wall, and steer around corners neatly.

8.7 Cinemachine Camera System

Cinemachine is a downloadable Unity package with multiple types of cameras available to implement [13]. We have used three different camera types: Virtual Camera, State-Driven Camera and Dolly Track with Cart.

The game starts with a pan across the city, whilst the gang boss introduces the aim of the game. This pan is done using a Dolly Track with Cart and Virtual Camera. The Dolly Track has been custom defined with seven waypoints around one edge of the city, and the speed at which the cart travels along it has also been custom defined. A virtual camera follows this cart along the track, whilst always looking at an empty `GameObject`, located in the center of the city.

At the end of the introduction, the camera blends to the main camera. This Virtual Camera, is used for the majority of the game and is set to `Follow` and `LookAt` the player's bird. When following the bird, the camera follows the Transposer algorithm, meaning it moves in a fixed relationship to the bird. To avoid the feeling of motion sickness when playing the game, the camera ignored all rotations with the exception of yaw. This meant the world never rotated whilst the player rotated left or right but did rotate when the player moved up or down. The rotation of the virtual camera was further defined by setting the `Aim` properties. We chose the Composer algorithm which ensures the camera is rotated to face the assigned `LookAt` object. Originally, we kept all the default settings for this algorithm as we simply wanted the camera to always look at the bird. However, it became apparent in user testing that users did not like that the city also moved when the bird was hovering. To overcome this, we increased the height of the dead zone to encompass the bird's height when hovering, meaning when the bird was hovering, the camera did not move. One of the extensions of the Cinemachine Virtual Camera that we added to the main camera is the Cinemachine Collider. This extension aims to always keep the `LookAt` target in sight. When this is not the case, the collider pulls the camera forwards until the target is in sight. Out of the three strategies available to overcome occlusions, we found pulling the camera forward was the most intuitive to the player.

As previously mentioned, each new round is introduced with a cutscene. This involves first panning to a Virtual Camera high in the sky looking down at the city, before blending to a camera at the respective location of the round introduction. The camera in the sky looks at an empty `GameObject` placed on the ground near the center of the city. We have set the `Aim` property to 'Hard Look At' which means the camera will remain looking directly at the `GameObject`. The Virtual Cameras at the locations of each round introduction all have the same `Body` and `Aim` properties, but respective `LookAt` targets. At the end of the cutscene, the player is panned back into the sky and then slowly panned back to the main camera looking at their bird. The decision behind using this panning approach was to give an establishing shot of the minibosses, allowing players to better locate

them within the city.

All of these transitions are managed by the Cinemachine Brain, which is responsible for configuring the Unity camera, and the Cinemachine State-Driven Camera, which is responsible for activating the correct Virtual Camera. This activation is done in script, and a change of Virtual Camera triggers a blend between the cameras. To the player this looks like a pan to and from the sky.

To give the player the effect of a smooth pan, we defined our own Custom Blends. This defines the style and duration of the blend. The blend from the main camera to the sky camera is shorter than the blend from the sky camera to the main camera. This signifies the start of a new cutscene, and at the end, gives the player some time to process what they had heard in the cutscene about what their next mission is. We chose to use the 'Ease In Out' style, which is an S-shaped curve. This results in a smooth transition, adding to the cinematic experience.

8.8 Visuals

Creating a vibrant and dynamic world requires strong front-end development. To do this, we used complex modelling techniques, multiple custom made shader graphs, particle effects and lighting. We decided to use the Universal Render Pipeline, as it supported our skybox lighting system, and allowed us to model effectively in the style we had chosen, with a lower performance cost than that of the High Definition Render Pipeline [14].

8.8.1 Research

We initiated our design process by collating reference images that matched the aesthetic we wanted to create. We produced a low-poly mood board, discussed the particular features we favoured, and used these as inspiration for our designs. We then drew up initial sketches of the city buildings, people and environment as shown in Figure 8.

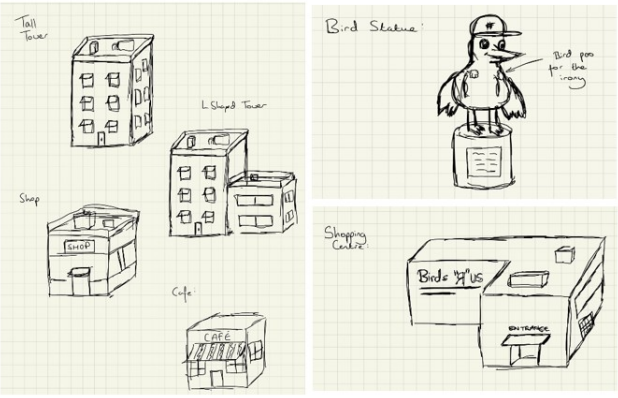


Figure 8: Initial sketches of city buildings, people and environment

We also formed a colour palette for all of our visuals. This included primary branding colours, as well as secondary and tertiary colours for both text and world models [15]. Modelling from reference was vital, and allowed us to confidently create our assets with a clear vision that could be easily referred back to by members of the team as we collectively developed. We then drew up our city blueprints, mapping out our roads and building blocks as an initial guide.

8.8.2 Modelling and animation

We used Maya as the primary platform to create all 52 of our unique assets, as it gave us a flexible and robust modelling toolset with which we could efficiently develop our dynamic world.

We began by blocking out our buildings to establish the correct scale when transferring models between Maya and Unity. We then created more detailed versions of these initial models, developing in a modular fashion to allow us to easily extend and alter our objects over time. This ensured our final world looked cohesive, despite it including so many different assets.

We used many different modelling techniques, these included extrusion, wedge chamfer and poke, and smoothing to name a few. Varying our processes allowed us to model much more efficiently and ensure we were constantly looking for better ways to develop. Our modelling process was iterative. After initial objects were created, we would conduct feedback sessions and continue to make changes until we were content with our final models.

Satisfied with our geometry, we needed to rig skeletons for the models that required animation (e.g. the mini-bosses and birds). This process involved creating IK Handles that would then allow us to focus on our objects goal positions, instead of getting bogged down in individual joint rotations. We initially experimented with binding skins to our skeletons, however preventing aspects of our models from deforming unnecessarily proved costly as it required far more joints to be created. Therefore, we decided to use parenting techniques to have joints control only specific parts of the geometry.

With our characters rigging systems set up, we could now animate efficiently. We set each models rest poses, making it easy to refer back to during our key framing process. This was particularly useful in regards to characters walking motions, as the rest position made up the downward pose the walking cycle.

We also made use of Unity's animation tools. Firstly, we created Animation Controllers that could dictate which phase of animation models would be in depending on world events. This can be seen as our birds switch between a flapping and gliding motion, dependent on their current pitch. Secondly, we developed animations

on the bank doors during the Robber cutscene, to give the effect of the Robber bursting through them as the alarm goes off. The Balloon cutscene also includes models of people hanging from the ropes as they fly away, with added variation by assigning the people types attached at random in scripts. The Mayor, Robber and Kid minibosses also include unique animations, from running to speech gestures, making them stand out against our general population and giving them realistic motions.

8.8.3 Assets

Below is a categorised breakdown of our final assets. A subset of these can be seen in Figures 9-12.

Buildings: We started by creating five core building types, comprising of small and large houses, tower blocks, L-shaped towers and shops. We varied the material assignment and scaling of these structures, to add levels of asymmetry and realism to the world. We then created our unique landmark buildings, such as town hall, bank, and shopping centres, creating variation and eye-catching features across the map.

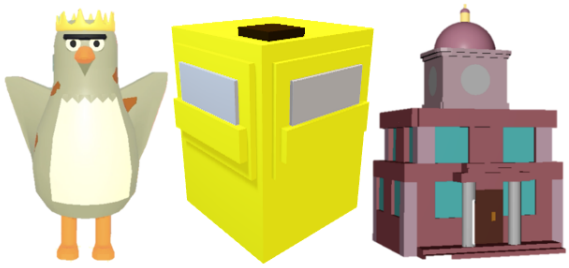


Figure 9: Bird statue and buildings

People and Birds: We developed six different humanoid characters in our world, and randomly spawn them in throughout the game, alongside the four miniboss models which generate during our cinematic cutscenes. Similarly, we developed two main bird models, for both our playable characters and the flocks.

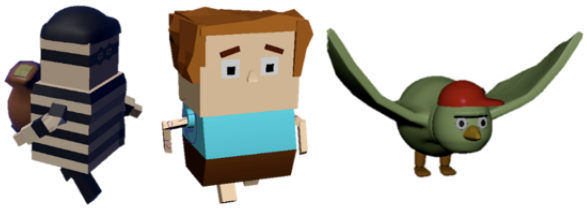


Figure 10: Robber, person and bird

Environment: Our city includes three main parks, and numerous other green areas. Filling each are seven variations of trees and bushes, as well as picnic benches and tables. Alongside this, there is a fountain that makes use of particle effects in the main park and a bird statue outside the town hall. The roads are also lined with beautifully designed lampposts which turn on as the day progresses. Our world also includes two beaches, with palm trees, boulders, and beach balls on them. Finally, we created clouds and the railings that line the canals.

Figure 11: Environment

Special Features: We also modelled a variety of props and decorations to make our cutscenes more engaging. The Robber round has a trail of money that follows this path. The Mayor round includes a stage, podium and bunting to make his speech backdrop eye-catching. The Balloon round centres around a beautiful carnival, featuring circus tents, rings and ticket booths.

Figure 12: Special decorations

Overall, the unique game aesthetic was noted as one of our strongest assets in user feedback, and continues to be one of our proudest achievements.

8.8.4 Shaders

Ocean shader: The effect of the ocean shader is shown in Figure 13. It is generated by moving two normal map images in opposite directions, creating the natural looking wave movement. The shader also has turquoise highlight added on top, giving a photo plankton effect. This is generated using a Voronoi noise node.

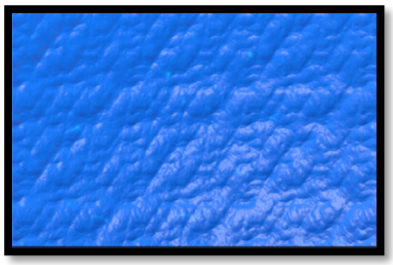


Figure 13: Ocean Shader output

Pavements and road shader: The pavement shader uses rounded rectangles and lerp nodes, before blending them. They also include multiple addition, multiplication, and random range nodes to implement further details.

Figure 14: Road and Pavements shader output

The road shader is made up of three separate sections: inner, outer, and middle lines. These are combined to form an easily extensible overall shader graph. Variable tiling and offset attributes can also be applied onto the rectangle nodes, to create multiple different effects, as shown in Figure 14.

Grass and beach shader: The grass shader comprises of multiplication and addition of a set of gradient noise nodes. Similarly, the beach shader uses addition and blending of sine and rotation nodes to add a 3D aspect to the beaches, as shown in Figure 15.

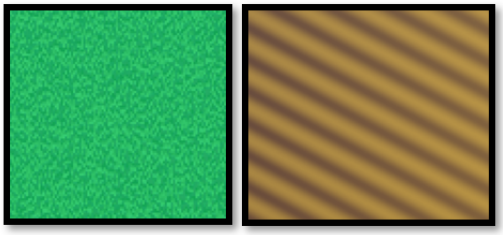


Figure 15: Grass and Beach Shader output

Interior shader: The interior mapping fragment shader shown in Figure 22b works by calculating a vector into the window from the camera, calculating that vector's intersection with an 'interior' cube, then sampling a cubemap with the resulting coordinates. For performance reasons, we use a single hand-authored cubemap for all buildings, rather than baking custom cubemaps for each.

8.8.5 Particle Effects

Splatter effect: When a projectile collides with an object in the scene it causes a splatter effect. The goal for this, was to mimic the style of over-the-top splashes seen in animated films and cartoons. When it collides, a collection of droplets are emitted in the direction of the normal, away from the objects surface. These droplets will then fall and impact the ground emitting another splatter particle that stains the ground for 10s. We use a texture atlas to efficiently store the various splatter textures we created. Having multiple possible splatter textures stops the effect looking repetitive. Using a texture atlas makes it easy to create additional splatter textures and include them in our world.

Money trail effect: When the robber is fleeing the bank, he leaves behind him a trail money falling out of his bag. This effect was achieved using a subemmiter where the primary emitter produces a mesh particle which then colliders with the floor producing a image of a note across the ground. In order to keep the money stationary on the floor run the simulation in world space rather than locally.

Fireworks: The firework particles are used to celebrate the end of the game. It is made from three particle effects and utilises the trail function. These components are plain rockets, bursting rockets and exploding rockets.

Fountain effects: The fountain particles beautify the main park in the residential area. It consists of a cone shape with a minute radius and angle to look as realistic as possible. It changes the size, colour, and force of the particles over its lifetime, creating a realistic water flow effect.

8.8.6 Prefabs

When handling assets at such scale, it was imperative to maintain structured prefabs to efficiently update and extend our world. This allowed us to organise our scene hierarchy, for example having just one prefab containing all lampposts in the world, instead of a single prefab for each object. We were then able to edit these assets with one attribute change, rather than having to make changes on an individual basis. We repeated the prefabbing process for all our world objects, including people and birds. We modified a script that was able to automate the process of making scene objects prefabs, which proved incredibly useful given the number of models in our world.

8.8.7 Lighting

Our game takes place over the course of a day. To accomplish this, we use a real-time lighting system. Our

scene contains three sources of light: environmental light from the skybox, global illumination, and directional light.

Figure 16: Before

Figure 17: After

The comparisons seen in Figures 16 and 17 show changes that happened when we added ambient occlusion on our game. This effect is subtle but makes models look less flat by simulation soft shadows that occur in creases or surfaces that are close together [16].

Figure 18 below shows the effect of specular highlights on our game. It causes the reflectiveness that appears on the water. We carefully set the smoothness on our all materials to make them believe and consistent with the style of our game.

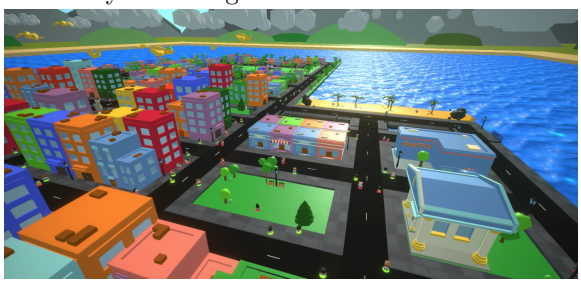


Figure 18: Specular highlights

As the game progresses day turns to night and the lampposts in our world switch on. How these are implemented is discussed in the performance optimisation (Section 8.11).

At night is also the best time to see the emissive materials we use. An example of this is the circle beneath the good and bad people, so that players are still able to easily locate bad people. Using emission allows us to highlight objects that otherwise would be difficult to see. In particular, we made our waypoints emissive beams that shoot up through the world, and with this glow effect they are eye-catching and easy for players to spot.

e

8.8.8 2D Artwork

When creating 2D artwork for our game we used the image processing software GIMP. While designing we added new layers for each individual component to make sure that images were easily editable. Below we explore the design choices made in some of these pieces of artwork.

Skybox: Creating a custom skybox (Figure 19) enabled us to match the background and environment lighting to the rest of the games aesthetic. Typically, skyboxes use a panoramic effect to warp the image such that it always looks spherical [17]. However, this was not appropriate for our cube-like theme. Our skybox design composed of a gradient effect behind layers of different types of hand drawn clouds.

Figure 19: Skybox

Gang Boss: Our Gang Boss design had two core inspirations: firstly the classic New York style gangster and secondly a 70's disco suit. We found the combination of these styles to be novel and entertaining. When creating the character of Gang Boss we wanted him to be wise but threatening, and felt that an owl would represent these characteristics. Using this brief we digitally drew the Gang Boss and added specific details like his pocket watch and hat.

Overlays: In keeping with the rest of our BirdGang branding, we created enticing title, loading, and credit scenes. For the title screen, we wanted to put across the storyline immediately. To do this, we incorporated a city skyline, with the main bird character stood at the forefront. With the loading screens, we continued the aesthetic, having graffiti as well as bird foot prints across the screen used in the menu. The in-game loading screens featured helpful tips for players while they loaded in. For the credit scene, we included all key characters and minibosses in the game. We also made the credit scene visually exciting, so as to encourage players to click to continue and resume playing in the open

world.

Splatter Particles: When designing our splatter particles we needed to create an easily reproducible process so that we could create varied particles. The process was as follows:

1. Create a white splatter shape
2. Apply a Gaussian filter to provide a blur effect
3. Threshold the blurred shape to achieve an outline effect
4. Add dark spots for detail

By using this process, we produced 6 different particles in the same style.

In-Game Canvas: One of the first things our players see when they load our game is our Heads Up Display (HUD). We designed this with the goal of having minimal distraction from gameplay while still providing all the information a player needs. An example of this design mindset is the semi-transparent background behind the text. This ensures that the text is still readable with anything behind it but is not obstructive.

At the top of our HUD, we have a progress bar which indicates the current round of the game. We added four round icons to the progress bar which represent the minibosses that the players will face. These indicators help set the pace for the game and ensure that players can see that new content is coming soon. We created a standard template so that if rounds were added or change, it is easy to adapt.

8.9 Audio

To add another dimension to our gameplay experience we utilised dynamic sound effects. Character centered sound effects include a poop splatter when the bird poops and a whoosh effect when it collides with an object.

Our world contains environmental sound effects that add another level of depth to the cutscene story telling. In the carnival cutscene, we introduce a background storm audio, bringing to life the story the Gang Boss is telling the players. During the carnival round we have included a 3D audio source centered in the circus that is triggered when players get close enough. The sound levels from the players left and right earphones differ for this audio, depending on the position of the bird relative to the audio source.

Throughout the course of the game the Gang Boss guides players through the events with high quality voiceovers, recorded on specialist equipment. We customised our audio cues to reflect the success (or failure) of the players after each round in the game. If players defeated a miniboss, they would be congratulated on their teamwork. If players failed to defeat a miniboss they would be informed of such failure. After the final round the Gang Boss will reflect on the performance of the players, and if they defeated enough minibosses, allow them to join his gang.

To create our soundtracks, we worked with two composers, Ollie Hatch and Zitian Zhao, from the music department at The University of Bristol. We met with both composers to explore our game and formulate a plan for the feel and structure of the music. From these meetings we received four soundtracks for the game and one soundtrack for the lobby. Each of the four in-game soundtracks are customised to reflect their assigned round. For example, the carnival soundtrack has more jazzy tones whereas the robber soundtrack has more mischievous tones.

Audio is managed using an audio manager which has a few advantages. After the initial set up, this method facilitates the simple editing and addition of new tracks. It also allows us to easily play audio in any part of the game, which makes it efficient to play dynamic voiceovers at the end of each round.

We also have a custom class for the sound which allows us to implement new editing options. We use two audio mixers: one for the music and one for the sound effects. This allows for differentiation between the two and allows the player to control the volume of each of these on the escape canvas.

8.10 Communication

Over the course of the game, players face four miniboss encounters. Each miniboss challenges the players' teamwork skills by requiring every player to have pooped on them to be defeated. To facilitate communication between the team members we implemented a system that incorporates three features: player names, player icons and player waypoints. A player's name can be entered in the main menu, and this can be seen by their teammates above the player's bird character in the game.

The player icon is also positioned above each player in the game and is used to indicate the location of each player to other players. Figure 20 shows an image of this. The position of the player icons on the screen is unique for each player. It factors in the location of the assigned player and what the local player can see. If a local player's teammate is out of their line of sight, the indicator remains at the edge of the screen closest to the location of the teammate. This allows for the local player to gauge a sense of where their teammate is so they can head in the correct direction to find them.

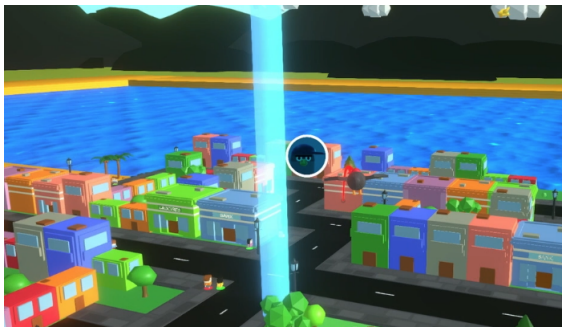


Figure 20: Waypoint and player icon

The waypoint, as shown in Figure 21, is a glowing beam that can be placed by pressing F. Its location is set to be the calling player's current location and an RPC event is triggered that tells each of the other player's systems to show this waypoint at that location. Alongside the placement of the waypoint text appears on the screen for every player, "A miniboss is near [calling player name]. Help them!", this draws the players attention to finding the waypoint and informs them of the reason the calling player is requesting help.

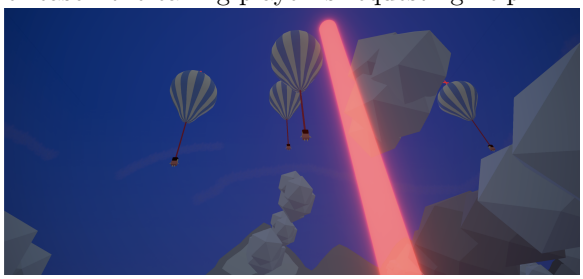


Figure 21: Waypoint

To group all these features together clearly and concisely we have colour coded them for each player. For example, a player with assigned colour red has a red icon, a red waypoint and red help text. This means that at a glance, players can identify who has called for help and their location.

8.11 Performance optimisations

As our world became larger and more detailed, we began experiencing a drop in frame rate. In this section we explain the various optimisations we have used.

To target our optimisation, we used the Unity Profiler. This indicated that the factor we were bound by was rendering, hence our optimisations are mainly focused on this.

8.11.1 Occlusion culling

By default, Unity uses frustrum culling where objects outside of the camera view are not rendered [18]. This however was not enough; we were still experiencing large drops in frame rate while in the city area of the map. To improve this we used occlusion culling. Occlusion culling improves performance by disabling the rendering of objects that are not visible when blocked by other objects. By dividing the world into 3D cells, Unity can pre-bake a map of which objects are visible from which cell. In-game, meshes are then selectively rendered depending on the camera's current cell [19]. This is particularly effective in the city area, where large buildings obscure much of the island.

8.11.2 Remodelling buildings and people

The large number of vertices and faces in our game were causing a decrease in performance. Due to having 200 people and 150 buildings which made up a significant proportion of this total, we decided to shift our focus to these. In the initial stages of our design process, we used techniques that were unknowingly adding a disproportionate amount of vertices than necessary. Each person and building had 3000 and 2500 vertices respectively. We found that creating sub-divisions and extrusions caused the largest increase in faces.

When remodelling the people and the buildings, we only sub-divided necessary faces as opposed to whole objects and implemented techniques such as multi-cuts rather than entire edge loops. This only created vertices at a specific location rather than around the whole model. Using these techniques, we successfully reduced the number of vertices to 250 per person and 500 per building. This resulted in a dramatic performance improvement, increasing the frame rate from approximately 10 FPS to 35 FPS, while still maintaining a visually appealing game.

8.11.3 Level of Detail

Due to the nature of our game, a large proportion of our map is often visible. This can put a strain on performance. To minimise this, we have used Level of Detail (LOD), where meshes are switched out for cheaper ones when they occupy a smaller portion of the screen [20].

Buildings are a good example of this. They are numerous, often not occluded, and contain transparent windows which are expensive to render. At one point profiling showed that transparency on mostly distant buildings was taking almost half of the rendering frametime, despite only counting for a fraction of the vertex count.

(a) LOD0 with interior geometry (b) LOD1 with interior mapped windows (c) LOD1 with opaque windows

Figure 22: Tower Block LODs

In our final implementation, all buildings have a single additional LOD which removes transparency by swapping the transparent window shaders for opaque ones, as well as removing interior geometry and other small details. We replace the transparency with either a cubemapped opaque shader, or, on more visible windows, such as those on tower blocks, a custom 'interior mapping' shader is used. This projects a cubemap onto the interior, giving an illusion of parallax without drawing additional geometry [21]. These methods are seen in Figure 22. Flocking birds in the distance also have simpler meshes, and animations are removed at a further LOD.

In addition, some LODs use normal maps, which improve shading quality on low-poly meshes by providing fragment shaders with higher-resolution surface normal data derived from the high-poly meshes [22].

8.11.4 Static and Dynamic batching

To further diagnose our performance issues, we used the Unity frame debugging tool. This allowed us to ascertain the number of draw calls being used to render a frame at any given time. A draw call in Unity is a call to the graphics API. This involves setting up the materials, textures and shaders needed to render the object in the frame. Although the draw call itself is relatively inexpensive, it is the set up that makes it costly[23][24].

Before our optimisations, we had approximately 5000 draw calls for each frame. When we stepped through the draw calls, we noticed that a lot of them were rendering the same material. This meant the graphics API had to do the same set up repeatedly. This was inefficient and therefore we turned our attention to static batching.

Static batching takes the vertices of numerous objects and transforms them into a combined mesh ready to be rendered [25]. This is more efficient as the materials, textures and shaders do not need to be set up as often, thus reducing the overhead that comes with a draw call. To implement static batching in our game, we marked all the objects in the scene that would not move as `static`. We also looked into using the same materials where possible, to further reduce the number of set ups needing to be performed. We found this worked well, however recognised there was still a significant number of objects in our scene that were not static and needed attention, such as the agents. This led us to looking into dynamic batching.

Dynamic batching works in a similar way to static batching; however the objects are transformed into one mesh on the CPU, rather than the GPU [26]. One way of implementing this is enabling GPU instancing on all materials applied to dynamic objects. We found dynamic batching was not suitable for our project as the overhead involved in GPU instancing materials can sometimes be worse than the improvements it gives, for example, when applied to objects with less than 256 vertices [27]. In our project although each object had a combined vertex count over 256, the specific mesh that each material was being applied to had less than 256 vertices.

Having implemented static batching, we reran the Unity frame debugging tool and found that we now had approximately 1000 draw calls.

8.11.5 Lamppost lighting

Initially, each lamppost had a spot light attached to it. Spot lights are computationally expensive, and as there are 95 lampposts in our game, this resulted in a large drop in frame rate. We replaced each spot light with a semi-transparent cone with an alpha gradient that decreases towards the ground. This created a similar effect, without the performance impact.

8.11.6 Flocking

One of the few scripts that impacted performance was the flocking script. To tackle this, we reduced the number of birds that used obstacle avoidance to only the loner birds. We then also reduced the number of `Physics Raycasts` that each of these birds did. Furthermore, we reduced the frequency of direction updates to the minimum possible before it became noticeable.

References

[1] B. W. Tuckman, "Developmental sequence in small groups.," *Psychological Bulletin*, vol. 63, no. 6, p. 384–399, 1965.

[2] Pim, "Feedback At Netflix: 4 Powerful Guidelines." https://corporate-rebels.com/feedback-at-netflix/, Apr 2022.

[3] scrum.org, "What is Scrum?." https://www.scrum.org/resources/what-is-scrum.

[4] B. Wirtz, "Unity vs. Godot: Performance, Community Support, Ease of Use, & Pricing." https://www.gamedesigning.org/engines/unity-vs-godot/, 2022. [Online; accessed 05-May-2022].

[5] M. Malhotra, "Unreal Engine vs Unity 3D Games Development: What to Choose?." https://www.valuecoders.com/blog/technology-and-apps/unreal-engine-vs-unity-3d-games-development/, 2017. [Online; accessed 05-May-2022].

[6] Unity, "About Unity Test Framework." https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html, 2021. [Online; accessed 17-March-2022].

[7] VRTogether, "Why And How To Automatize Testing In Unity Projects?." https://vrtogether.eu/2020/11/23/automatize-testing-unity-projects/, 2020. [Online; accessed 04-May-2022].

[8] rutter, "When to use Quaternion vs Euler Angles?." https://answers.unity.com/questions/765683/when-to-use-quaternion-vs-euler-angles.html, 2014. [Online; accessed 10-February-2022].

[9] R. Sudhamani and K. Merriliance, "Deformation exploration in mass spring model using euler and verlet integration methods," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 2, p. 119–124, 2019.

[10] Unity, "Inner Workings of the Navigation System." https://docs.unity3d.com/Manual/nav-InnerWorkings.html, 2022. [Online; accessed 05-May-2022].

[11] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*, 1987.

[12] C. Reynolds, "Not Bumping Into Things." https://www.red3d.com/cwr/nobump/nobump.html. [Online; accessed 05-May-2022].

[13] Unity, "Using Cinemachine." https://docs.unity3d.com/Packages/com.unity.cinemachine@2.8/manual/CinemachineUsing.html, 2022. [Online; accessed 02-May-2022].

[14] Unity, "Render pipelines." https://docs.unity3d.com/2019.3/Documentation/Manual/render-pipelines.html, 2022. [Online; accessed 02-May-2022].

[15] "BirdGang Design Reference." https://docs.google.com/document/d/1bjHugwP0o2oHm3qYfe2ZLHRoiDBjEhEDx4OUXIIc89E/edit.

[16] L. Bavoil and M. Sainz, "Screen Space Ambient Occlusion." https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf, 2008. [Online; accessed 01-May-2022].

[17] Learn OpenGL, "Cubemaps." https://learnopengl.com/Advanced-OpenGL/Cubemaps, 2022. [Online; accessed 05-May-2022].

[18] Unity, "Understanding the View Frustum." https://docs.unity3d.com/Manual/UnderstandingFrustum.html, 2022. [Online; accessed 01-May-2022].

[19] Umbra Software, "Next Generation Occlusion Culling." https://www.gamedeveloper.com/programming/sponsored-feature-next-generation-occlusion-culling, 2012. [Online; accessed 05-May-2022].

[20] Unity, "Level of Detail (LOD) for meshes." https://docs.unity3d.com/Manual/LevelOfDetail.html, 2021. [Online; accessed 04-May-2022].

[21] E. Persson, "Interior Mapping." https://www.humus.name/index.php?page=3D&ID=80, 2008. [Online; accessed 05-May-2022].

[22] Learn OpenGL, "Normal Mapping." https://learnopengl.com/Advanced-Lighting/Normal-Mapping, 2022. [Online; accessed 03-May-2022].

[23] Unity, "Optimizing draw calls." https://docs.unity3d.com/2022.2/Documentation/Manual/optimizing-draw-calls.html, 2022. [Online; accessed 05-May-2022].

[24] Emscripten Contributors, "Optimizing WebGL." https://emscripten.org/docs/optimizing/Optimizing-WebGL.html, 2022. [Online; accessed 04-May-2022].

[25] Unity, "Static Batching." https://docs.unity3d.com/Manual/static-batching.html, 2022. [Online; accessed 04-May-2022].

[26] Unity, "Dynamic Batching." https://docs.unity3d.com/Manual/dynamic-batching.html, 2022. [Online; accessed 04-May-2022].

[27] Unity, "GPU instancing." https://docs.unity3d.com/Manual/GPUInstancing.html, 2022. [Online; accessed 02-May-2022].