University of BRISTOL

NAUMACHIA

Team “Szechuan Sauce” C

Angus Parsonson
Charlie Figuero
James Keen
James Lim
Rafael d’Arce
Yash Agarwal

Contents

Overall Individual Contributions

Team Member	Contribution	Member Signature
Angus Parsonson	1	
Charlie Figuero	1	
James Keen	1	
James Lim	1	
Rafael d’Arce	1	
Yash Agarwal	1	

Top Ten Team Contributions

1. Using Unity 3D, we built an immersive and competitive VR action experience - despite no previous experience with such technology.

2. Wrote over 2200 lines of C# code using an agile development method.

3. Extensive use of version control, continuous integration and sprint planning through GitLab

4. Boasts a robust and scalable enemy AI system.

5. Implemented responsive voice control commands.

6. Immersive and responsive custom hardware controls.

7. Later versions of the game only use 3D models, textures and animations that we designed/created ourselves.

8. Supports 2-player reliable multiplayer over both LAN and the Internet.

9. Has an original soundtrack.

10. Successfully responded to feedback from MVP testing to improve the gamers experience of the Beta.

Categorised Team Contributions

Professional Team Communication and Problem Resolution:

- Regular communication and meeting planning via a Facebook group chat.
- Regular pair and group programming sessions using screen sharing and voice chat on a Discord server.
- Assigned and completed tasks for all members of the group in one/two week intervals (sprints) using the task board on Gitlab.

Technical Depth and Technical Understanding Gained;

- Knowledge of networking (using the unity networking library for both LAN games and games to be played over the internet from different networks).
- Understanding of Unity.
- Creating hardware (the wheel).
- VR technology (including using unity controllers and our own custom controllers).
- Voice Recognition.

Technical Implementations, Solutions and Achievements Delivered:

- Created a bespoke wheel.
- Implemented voice controls.
- Both VRs work in multiplayer.

Professional Development and Testing Cycle using Professional Tools;

- Sprints tracked on GitLab.
- Branched for features and merged back into “dev” branch.
- Separated dev branch and master (release branch).

Game Playability, Interactivity and Controls:

- Fully playable
- Controls:
    - Steer using wheel
    - Aim using VR
    - Fire/Reload using Voice control

Look & Feel, Graphics Made/Generated, and Physical Game:

- All models modelled by the team
- Cartoonish vibe

Game Novelty and Unique Product Aspects:

- Bespoke wheel
- Novel use of voice control contributes to a more lively gameplay experience

Abstract

Naumachia is a virtual reality, arcade, battle arena game between two opposing players. Each player controls the captain of a naval ship. Player ships are equipped with cannons on either side of the deck, which can be fired and reloaded using voice controls. Additionally, each player must steer their ship using a physical wheel built for this game. Player ships always move forwards without any player input. Other non-player ships are also present in the game which attack the players with cannonballs. All ships have a health meter and can be damaged by enemy ships. Player ships are larger than AI ships, making them easily distinguishable.

Naumachia takes its name from the very real naval arena battles that the Ancient Romans used to stage. They would flood gladiator arenas and bring in real ships to battle each other. The visual design of this game takes much influence from these historical events.

Ancient Roman Naumachia.

The aim of the game is to gain a higher score than the opposing player by the end of the round. The default duration of a round is 10 minutes. Points are gained by eliminating other ships, that being AI or player ships. Enemy ships can be damaged and destroyed by successful hits with cannonballs or by being rammed by player ships. AI ships grant a small amount of points and increase the score multiplier when killed. These ships respawn frequently and indefinitely so there will always be AI ships present in the arena. If an AI ship which has been attacked by both players dies, the score and score multiplier is gained by the player which delivered the final blow to the ship. Player ships grant a large amount of points when killed and respawn with full health after a short delay. When a player dies, they keep their total score so far, but their score multiplier is reset to 1.0x. The effect of a ship being damaged can be seen by fires which appear as its health becomes lower, as well as floating numbers indicating the amount of damage dealt or received. When a ship is destroyed, it shatters, leaving behind wreckage that disappears after a short delay.

Players can aim their cannons by looking at the side of the ship towards which they want to fire the cannons. A visual then appears representing the arc the cannonballs will follow when fired. The cannons can then be fired with the voice command “Fire!”. Since the cannons are on the sides, players cannot shoot directly forwards or backwards. Additionally, player cannons have a limited amount of shots that can be fired in quick succession and must be reloaded with the voice command “Reload!”. AI ships have only 2 cannons, one at the back and one at the front. These cannons have full range of motion and will be fired towards players when in range.

The battle takes place within a Roman coliseum with various isles and obstacles positioned inside the coliseum. These obstacles can block fired cannonballs and impede ship movement in the arena. If a ship collides with the environment, it will take damage. Around the arena, there are 12 gates from which player and AI ships will spawn throughout the battle. Furthermore, there are different power ups scattered throughout the arena which can be picked up by player ships to boost the efficacy of the players for a limited time. These powerups include:

- Increased movement speed.
- Increased rate of fire for cannons.
- Increased score multipliers.

Additionally, there is a health pickup which restores health to player ships. Health pickups cannot increase the player’s health beyond their maximum health. These power ups respawn a few moments after their effects have expired.

Traditional UI elements such as score and health points are represented in in-game objects to further increase immersion. The score is represented on a paper scroll positioned next to the helm and the amount of health remaining can be seen by the amount of fires on the ship.

The inclusion of voice controls, a physical steering wheel and the integration of virtual reality make Naumachia an exciting, immersive, and competitive game which can be understood and enjoyed in short playing sessions.

The Team Process and Project Planning

How the group worked:

- Overall we had an impressively collaborative and transparent team dynamic. Our team worked well together throughout the development of our game, from the initial idea generation sessions to the final late-night bug squashing rampages.
- We followed an agile development methodology: we had a backlog of tasks that we would need to get done for a release, every sprint we each take on some

What went well:

- We enjoyed working with each other
- Communication was strong
- Strength based allocation
- Organised ourselves into sub-teams to deliver features across full stack concurrently.
- Good version control practices
- Great backlog and sprint organisation.
- Reverb thinking

What didn't work:

- In the development of our MVP, we did not integrate as often as we should have. This caused some last minute problems in our demo due to different components breaking each other.

Methods employed to assist collaborative development:

- GitLab issue boards allowed us to keep track of our own and other team members' tasks, as well as development workflows. This allowed all team members to see what other members were working on; helping keep everyone connected and build team solidarity since everyone could see everyone else's progress. ito
- The development workflows were especially useful to help without collaborative work since we could clearly see what needed to be done before another task could be attempted. This encouraged our team to keep up active development consistently so as to not delay other team member's from being able to complete their tasks.
- Where there were large-scale and lengthy tasks to complete we assigned multiple people to the issues. This was evident, for example, with networking, where the task was almost a constant issue that would be prevalent throughout almost all of the game's development since almost all elements of the game would require some amount of configuration to work over the network.
- Meeting as regularly as desired was sometimes difficult because of schedule conflicts due to members of the team taking wildly different electives. Once again good version control was an effective vaccine here, as it allowed work to be shared remotely. VSCode's live-share functionality was also an effective method of pair-programming when in-person meetings became difficult during the start of the coronavirus pandemic.

What we learnt about collaborative team work:

- Pair programming could be much more efficient
- By keeping all team members up to date and clued in on what was left to be done (eg. through gitlab boards), we could cut back on wasted/duplicated work.
- After some detrimental commits it quickly became apparent that the team needed a more structured procedure to change the shared repository. We began created a master branch for releases, then a dev branch which was for working versions in development. After a feature/ bug was added to the board, whoever was assigned to the task pulled from the dev branch to develop the change. When the changes were finalised a pull request was created and another team member would review the change before finally merging back into dev. After we implemented this change time wasted on bad commits was drastically reduced.

Would you change anything about the way you worked if you had to start again:

- Integrate early and often.
- Have modular designed AI from the start.
- More rapid prototyping of the game.
- Learn how to use electronics to build our own sensors.

Planning and time-management process worked during development:

- We would have two week sprints: everyone would be assigned issue tickets on the Gitlab board that would take roughly 20 hours. We would communicate almost daily on our group chat, and meet at least once a week to work together in person.
- At the end of the sprint, we would perform integration. We also review the backlog and add any new issues that have arisen.

How our plan changed:

- Naumachia undertook a massive transformation after the beta build game demo. Our original idea for the game was a cooperative experience where one player acted as the captain, steering the ship and firing cannon balls, while the second, playing as the First Mate experienced the game from a first person perspective. Tilo suggested that we change to a player vs player concept and after considering this we determined that there were a number of advantages to this approach:
- Reduced workload. As the previous game idea had asymmetrical gameplay, almost twice as much content had to be developed as we were essentially creating two different games- one for the captain and one for the first mate. By just focusing on one of the aspects we could free up time and therefore deliver a better experience for just one side of gameplay.
- Decreased complexity for networking. Having a VR player on a moving ship was showing itself to be more of a challenge than we had originally anticipated in terms of networking. The networking library we used, UNet, was depreciated and had a few gaping holes in its functionality which exposed themselves as we tried to fulfil our tasks. Streaming the relationship between the Captains position and the First Mate's was an incredibly complex task when attempting to create a lagless experience.

Switching to 1v1 gameplay substantially decreased the difficulty of networking throughout the project and dodged any issues with UNet.

- Attempting to create engaging gameplay. The original gameplay attempted to lean on and emphasize the cooperative aspects of the idea to create the most enjoyable experience for the player. However, after many brainstorming sessions and failed attempts, ideas on how to implement this in practicality dried up. Switching to versus gameplay completely changed the dynamic between the two players for the better. Although there was no more cooperation between the two parties, they actually ended up interacting more with this change to the concept as gameplay revolved around shooting at the other player, predicting their moves for strategic advantages and beating them to the chase for power-ups. Additionally, the requirement for engaging AI could now be slackened as the main challenge would come from the other players actions and would not have to be coded into the game.

Individual Contributions

Angus Parsonson

- **In game scoring system:** Devised and coded a full scoring system to work for both players so they could see their own score and the other player's score. (5 hours)
- **Designed and added waiting room functionality and main menu HUD:** Wrote instructions for each player and made sure neither sailed into the arena until they both clicked ready. (10 hours)
- **Developed player spawning system:** Made sure both players returned to their spawn point upon death. (5 hours)
- **Added global time system:** For both players to see and to make sure the time limit of ten minutes could signal the end of the game. (5 hours)
- **Helped implement a new cannonball firing system**: Along with other team members helped change the firing system to one which felt better and was more fun for the players. (10 hours)
- **Added music to the start menu and the main game loop:** Put the music composed for us by Will into the game. This included a menu loop, a main game loop and victory and defeat music. (10 hours)
- **Undertook team leader responsibilities:** This included making sure the team was communicating and collaborating effectively, organising meetings with our mentors and drafting/sending emails to Tilo etc. (10 hours)

Charlie Figuero

Added Physics-based collisions between players, enemies and the arena (20 hours)
- We encountered a large bug that caused player ships to move through objects that should pose as obstacles. This was due to an early design decision to move player objects by updating the position manually rather than through forces applied using the Physics engine.
- I was able to refactor the player movement scripts to work using the Physics engine, despite it being quite late into the project.

Organisation Responsibilities (20 hours)
- Taking notes during meetings.
- Creating issues on Gitlab.

Set-up suitable version control for Unity projects (4 hours)
- Researched and added the appropriate .gitignore files and set up Git LFS.

Built Respawn System (20 hours):
- Developed an efficient system to respawn both player and enemy AI ships that efficiently avoid deleting and recreating the same objects.

Angus Parsonson and Charlie Figuero (*Networking Team*)

- **Networking for both LAN gameplay and gameplay over the internet:** Our game ran with two Oculus Rifts and two gaming PCs which needed to be networked together. Our plan for games day was to connect them via an ethernet cable. In order to accomplish this all of the code had to be written with networking in mind. At the start we had to rewrite a lot of code from the different aspects of the game in order for the networking to work, but as we progressed - everyone in the team started to understand how to network their code. Often we would only have to change a small amount of functionality or add a few lines of code in order for it to operate correctly. In addition to this only certain things had to be networked for the game to run smoothly, we had the attitude that the more that could be run on each individual client, the better.

  After our whole team left Bristol for quarantine. We had to change the code to run over the internet rather than on LAN. The underlying code was the same however there was some work to do with Unity. The majority of our work was spent on the Networking as this was a major (changes needed to be made) and important task. (100 hours each)

James Keen

- **Initial development of core mechanics:** Identified the core mechanics that would need to be built for the initial iteration of the game. This included development of mechanics for the initial player's ship, a firing system, a point system, and a basic enemy AI. I started with the initial empty scene in Unity that we had set up as a team and set about building the base game.
    - Watched tutorials on game development so that the project would be given a solid foundation. This ensured that good game development practises were implemented, including modular scripting. This meant that the game was easily expandable for when other members of the team wanted to add features.
    - Added documentation for all developed features onto our gitlab project so that all members of the team could easily get to grips with the initial features and objects within our game.
    - ~20 Hours

- **Helped set up git project:** Collaborated on creating git init to prevent unnecessary unity files from being uploaded and set up git lfs:
    - ~5 Hours

- **Iterative development of complex firing system:** Expanded on the initial rudimentary cannon firing system that I had developed for the base game.
    - Conducted research on the best way to innovatively harness the technology that we had (2 Oculus Rift devices), conducted user experience tests and

developed numerous iterations of different methods of aiming so as to develop the most user friendly system.
    - ~40 Hours

- **Powerups:** Implemented power ups that could dynamically appear, re-appear and have different effects on the players.
    - Utilised time control and had to work alongside the Networking Team to solve the many networking challenges involved
    - ~20 Hours

- **End game screen:** Created end game screen which displayed scoring information and important stats.
    - Utilised camera movement with care taken to avoid motion sickness.
    - ~10 Hours

James Lim

- **Pre-MVP First mate Oculus VR character controller:** Initially, I used the provided character controller prefab from the Oculus Integration package. However, after finding out this prefab was not compatible with moving platforms (which was essential for the player to be on the moving boat), I created an Oculus VR character controller from the ground up. This required a large amount of research on how the Oculus Rift worked with Unity, especially for controller input and the tracking of the camera and Touch controllers.

  Another feature the first mate was required to have was the ability to pick up objects with its left hand. Using a script from the Oculus Integration package for unity, I was able to include this feature into the character controller I had previously built. Additionally, I included a hand from the Oculus Integration package that would animate accordingly when objects were picked up.

  On the right hand, the first mate was equipped with a gun that could take out enemies. To implement this, I did some research into how ray casts worked in unity and determining if they collide with an object. To increase the immersive feel of the gun, I also researched player feedback in the form of controller vibration and sound. These features were essential to implement together as the method I used for controller vibration varied according to a respective sound effect.

  Lastly, I began implementing cuboid versions of enemies that would die when shot. This required research into instantiation of objects in unity to spawn the enemies.

  I estimate I spent 50 hours for the research and implementation of these tasks.

- **Modelling in Maya:** Applying the knowledge and skills acquired from the Character and Set Design unit, I modelled and created materials for the player and AI ships in a low-poly style that would not stress the unity engine, as high performance is essential for VR games. Additionally, I modelled and textured the ships’ cannons and scroll stand, as well as clutter to be positioned around the ships. This included: barrels, crates, buckets, cannonballs, and a treasure chest. I also modelled and textured the grabbable power ups scattered around the arena.

I estimate the total time spent modelling was around 60 hours.

- **Fire particles in Unity:** I experimented with both the Unity Particle System as well as the Unity Shader Graph to create visible fires for the game. We decided to use the ones created with the Unity Particle System as the ones created with Unity Shader Graph required our project to run with the High-Performance Rendering Pipeline. At this stage in our project, it was too late to change to this rendering pipeline.

  Once the fires were created, I also built a Fire Manager in unity that would spawn or despawn fires on ships depending on the health percentage of that ship.

  This task took around 20 hours to complete.

- **Ship shattering and despawning:** I modified the models I had previously created to include a shattered version in which each fragment of the ship reacted to the physics in the Unity Engine. I also created a small script to handle the event of a ship shattering, leaving behind debris that would despawn after a short delay.

  I spent around 15 hours completing this task.

Rafael d’Arce

Developed robust and scalable enemy AI (60 hours)
- It was required to re-implement the enemy AI after our MVP demo. This was due to hard-coded behaviour and it being difficult to tell what the AI was planning to do.
- The new AI system was built in a modular fashion making it easy to add new features. Also, it was designed based on Finite State Machines - making it easy to interpret what state of behaviour each AI is in.

Undertook DevOps and Lead Programmer responsibilities (30 hours)
- DevOps role involved organising and maintaining the Gitlab repository, testing and approving Pull Requests, setting up CI and ensuring that Games Day PCs had all necessary software installed.
- Lead Programmer role involved being familiar with the entire stack being used, helping organise sprints and ensuring feature integration went smoothly.

Worked with a music composer to create an original score (5 hours)
- Had the pleasure of working with Will Farmer, a composer, in creating an original score for our game. Will was interested in our project for its setting and concept, while we were interested in working with him given his previous experience with composing for video games.
- I was Will’s point of contact for our team and I communicated with him regularly to express our requirements for the score and to give feedback on his progress.

- Due to COVID-19, only a main theme, a menu ambient track and victory/loss stings were finished. However, there were plans for Will and I to create sound effects - including one idea to record maritime audio from the Bristol Harbour.

Designed and built a custom input device (50 hours)
- Had to learn how to use CAD software, Laser Cutters and 3D Printers to iteratively design a cheap yet very effective ship wheel (a helm) controller.
- Feedback from our MVP demo was clear that the prototype wheel greatly improved the game experience over the normal VR controls.
- Its weight, feel, and responsiveness is great for immersion.

Animated the ocean model (10 hours)
- The ocean model and its waves are based on a cloth/animator, resulting in smoother waves than a more realistic water simulation. This was done in hope to reduce VR sickness. Also, the game was tested with some non-developer individuals to ensure the moving water was not uncomfortable.

Implemented in-game voice commands (5 hours)
- A simple solution that makes use of the powerful speech recognition technology available in Windows 10.
- Allows users to make responsive commands - such as "Fire!" and "Reload" - adding even more to the great immersion of our game.

Yash Agarwal

Initial development of Player ship movement (MVP): Laid solid foundations on how the player ship moves. Watched video tutorials on how unity works.

- Wrote the initial movement script that was used for the MVP. The script controls the speed, rotation and the tilt of the ship and provides minimal motion sickness.
- Calibrated the custom wheel device with game code to get a smooth operation of steering the ship with the wheel.
- (20 hrs)

Initial development of Enemy-AI movement (MVP): created a basic working version of the enemy-AI for the MVP which needed some work and was later re-implemented.

- Coded mechanisms in place to follow the player ship.
- Coded the ability of the enemy ship encircles the player ship to get in optimal positions to fire.
- (10 hrs)

3D Modelling in Maya : With some previous experience in 3d modelling, along with hours of research and video tutorials I was able to model and texture the assets which transformed the scenes from the testing phase into resembling a decent game. A low-poly approach was followed to put minimal stress on the unity engine without sacrificing the aesthetic appeal.

- Colosseum (90 hrs)
- Volcano Terrain (25 hrs)
- Greek Column and Arches (40 hrs)

Software

How to run the game

Running our game requires two Oculus Rift VR kits, two custom ship wheel controllers and two Windows PCs. Unfortunately, due to the COVID-19 pandemic, we were unable to finish two usable controllers. Only one functioning prototype exists at the moment. The PCs are required to meet the minimum specifications to run an Oculus Rift (see https://support.oculus.com/248749509016567/) and have installed Unity 3D version 2018.4.16f1+ (specific versions can be installed through the Unity Hub app).

Ensure you have the VR set up before running the game - we recommend that you play sitting on a chair with the wheel controller on a desk. Also ensure that you have your networking set up before the game. If playing together locally, ensure both PCs are connected to the same private WiFi. If playing over the internet, ensure the Client player has the correct IP address to connect to

You can build and run the game from inside the Unity editor (Ctrl+B) or - if you already have built the project once - can run directly from the executable file.

One player must host the game: they should launch their game first and select the appropriate hosting option. The other player must either connect locally by entering as a client to the default port, or over the internet using the host IP address. Both players must ready up with the voice commend “Ready” for the game to start.

Adding new functionality

To add more enemy behaviour is simple - the AI system was designed in a modular fashion. Enemies are controlled by two scripts:

- *AIEnemy.cs* controls movement speed and navigation.
- *AIStateMachine.cs* controls the FSM-like behaviour by tracking possible states and the current state. The states are modelled by *AIState.cs* class.
  - An AIState has a public method *nextState( )* which, given the current state of the game (e.g. player health, enemy health, distance to player), it will return another AIState.
  - At every update call of the AIStateMachine, it calls nextState on the current state and transitions to the returned state.

Simply create new state classes by:

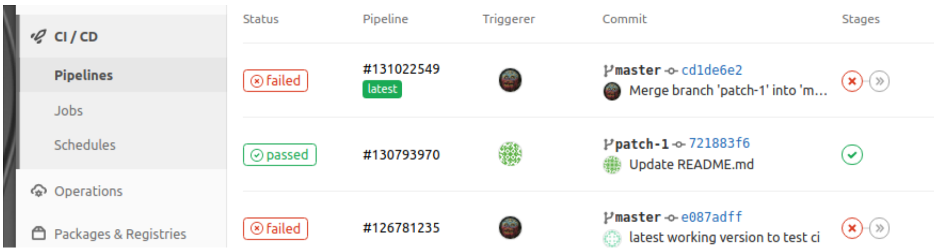
- Inheriting from AIState.cs,
- Overriding the nextState function with desired behaviour,
- Adding the type of this new state class to the AIStateMachine set of states.

Software maintenance during development

Gitlab was crucial for the maintenance of our software throughout development, thanks to the variety of free services it provides for developers. Most importantly - Git version control. Unity 3D projects, on their own, are often tricky to perform version control on - due to large media files and the many automatically generated meta files. These issues were solved with an extensive *.gitignore* file and the Git LFS (Large File Storage) extension automatically available on Gitlab.

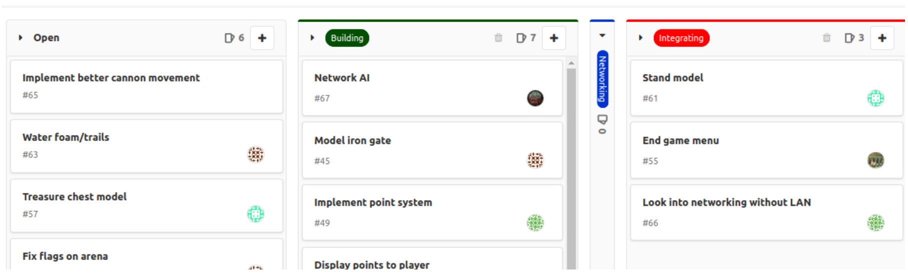
Our project repository has two main branches: Master and Dev. Master would hold the most recent polished release (e.g. MVP, Beta versions) whereas Dev would be unpolished but have any recently integrated features. No team member - other than the Lead Programmer - had the rights to push to the Master and Dev. Our strategy was for each member to branch out from Dev when they needed to implement a new feature. When they were finished implementing, they would create a Pull Request back into Dev. The Lead Programmer would test the integration by running the request branch on Games Day set up (most development was done remotely). Only if it passed this, would the pull request be merged into dev.

To ensure our Master branch is always working, we made use of Gitlabs in-built CI functionality. A simple Unity environment was set up to test automatically building and running the game.



Gitlab CI pipelines at work

Gitlab also provides a Kanban-like board for organising issues put up on our repo. It was invaluable in the organisation of our development - making it very quick to see what tasks one has to do, what tasks have been recently completed by others, and what can be started on next.



Gitlab Kanban-like board

Technical Content

Networking

The two main networking challenges for the project were finding and learning a suitable networking library for use with Unity, then secondly, choosing which components of the game were necessary to network, and which could be ignored without being noticeable to the players (in order to reduce network throughput and therefore lag.)

We used Unity's Network Manager to implement our multiplayer functionality. This uses UDP by default enabling a low latency experience. The downside to this UNet is that it is a depreciated library, meaning it lacks support in some areas and has design issues. However, after researching the alternatives (eg. Photon), we decided the gains to be made in converting our project were not worthwhile and that all use cases could be fulfilled with the original choice.

Gameplay consists of three dynamic elements: the two players, enemyAI actions and map wide events.

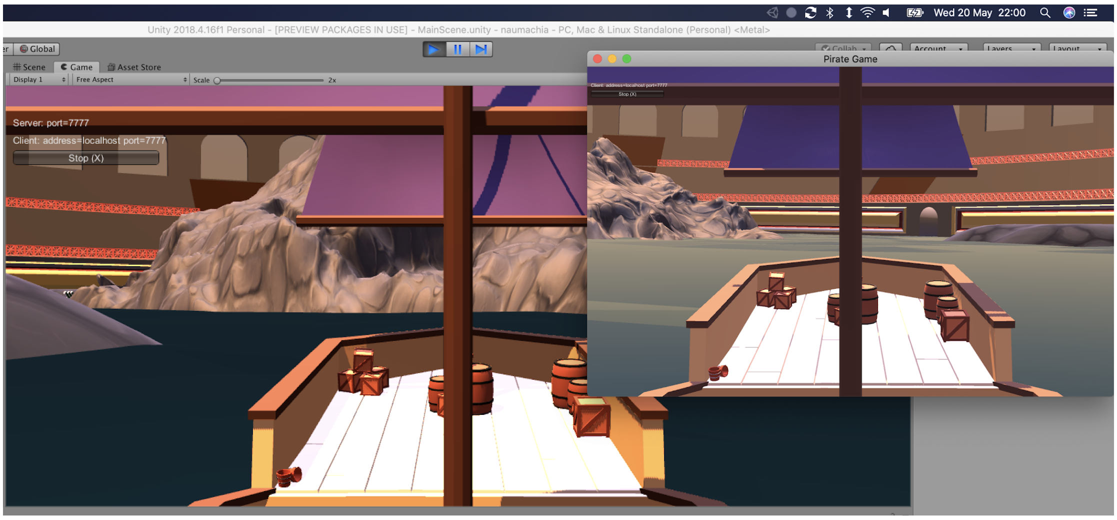
Map events were the least network intensive of the three elements: after broadcasting that the event is happening in a single flag, no other detail must be passed through the network- meaning a simple, low-latency solution.

Networking player actions played out as being the most network intensive of all networking action. This was necessary as any player action must be instantly communicated to the other to create a seamless multiplayer experience. We steamed all player actions using network transforms and networked variables. These streamed at roughly the same rate as screen refresh creating the illusion of instant response times.

Enemy movements were the most complicated part of networking. The obvious option was to stream the AI actions in the same way the player actions were streamed. However as there were many enemies this would cause a network overload. The solution was to create a state machine system where only changes in state were communicated across the network rather than streaming frame by frame positions and actions. This was possible as enemy action scripts/ai was determined by player positions and actions which are already available.

As this project was focused around the final games day event, special effort was put into networking the two computers around the MVB's network. We initially had difficulty in streaming across the network due to external security constraints, so our final solution was to log into user profiles through eduroam then disconnect and directly connect the two computers which ran the game to avoid the universities restrictions. This lan connection helped provide a low-latency experience.

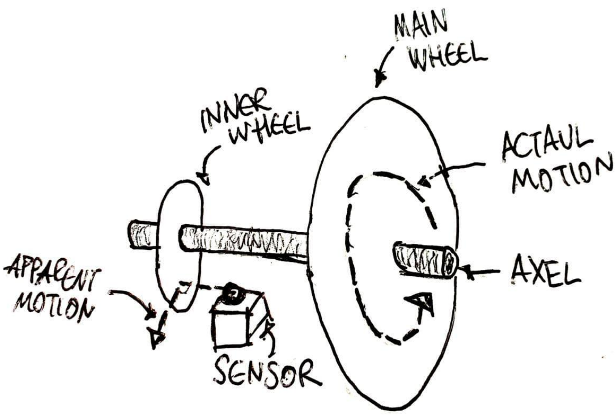
Below is a picture of two applications running on the same computer. One of the applications runs as a client and the other as a host (server and client at the same time). This functionality was possible by setting up a lan connection between the two applications, and made testing drastically more efficient by bypassing the majority of the set-up. By running different roles in the unity editor it was possible to debug both experiences.



Two players in the same game

Wheel Controller

Noone in the team had previous experience in electronics or Computer Aided Design and Manufacturing (CAD/CAM). We also had a small budget. The controller was an essential part of the game that needed to be integrated early, so rapid prototyping was required and we would have to find cheap solutions. We had an idea to use two wheels attached to a stand: one main wheel outside the stand for users to turn, and a smaller, inner wheel inside the stand. The idea was that if we had some sensor fixed in parallel to one point on the inner wheel, we could measure movement along an axis as the inner wheel turns with the main wheel.

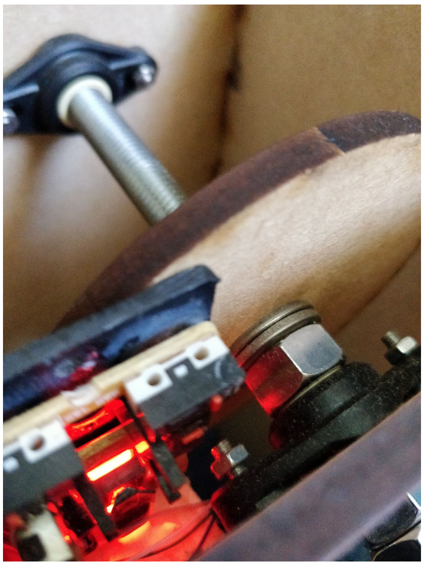


Sketch of idea for wheel mechanics

First, we built a proof of concept for this idea. It made use of a paper plate (main wheel), an old CD (inner wheel), a cardboard box (stand), a kitchen roll tube (axel) and the optical sensor of an old USB mouse. The stripped-apart mouse worked well so - when we were unable to be inducted into the electronics area of the Hackspace in time for our MVP - we decided to continue using it as our sensor rather making one from scratch.

Then, once we had access to the Hackspace, the next prototypes were built in wood. Experience was gained in using Laser Cutters (and their design software) to build the stand, and 3D printers (and their software) to produce a cuskjtom piece required to attach the main wheel to the axel.

Rather than trying to build our own model helm (ship wheel) - given our little time and no experience in detailed woodwork - we chose to buy a scale replica. This decision turned out to be very good as the textured feel and heavy weight of the wheel felt great to play with. A lot of effort was then spent in making the wheel's control feel responsive in-game (like you were actually controlling the ship). Most of those who tested our MVP demo noted that the replica wheel and its responsiveness created great immersion for them.

Left: Second prototype, Right: Mouse sensor at work

Internal mechanism

Unfortunately, due to COVID-19, we were unable to finalise the design and build two final controllers. Only one working prototype exists. Also, we were unable to complete the final task of using 3D scanning technology to produce a realistic digital rendering of the wheel. It would have been used in-game and moved in sync with the real world wheel to add even more immersion.

Enemy AI

The First AI system implemented was very problematic. It hard-coded all decision making into a single script. During gameplay, an enemies behaviour was uninterpretable during gameplay and the performance did not scale well with multiple enemies at once.

After our MVP, it was decided to completely rebuild our AI on top of Unity's in-built NavMesh library. This AI library allows developers to quickly add movement to their characters. A NavMeshAgent component will automatically calculate a shortest path (with A*) and move to a specified destination. A NavMeshSurface defines where this Agent can go, and NavMeshObstacles can be added to the Surface which the Agent will avoid.

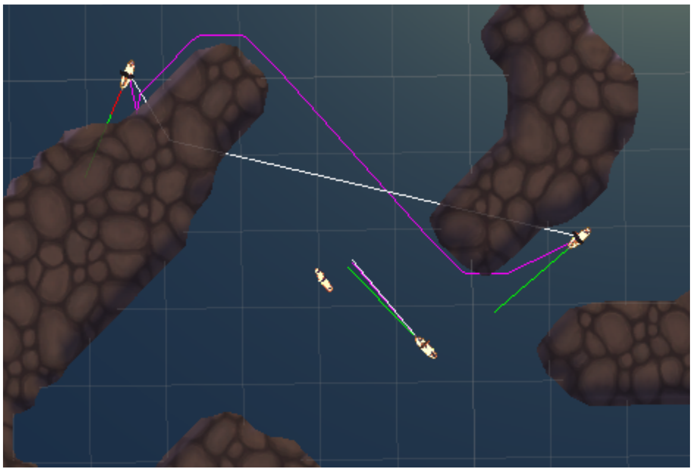
While very useful and scalable - thanks to the ability to pre-calculate and bake shortest path navigation to a surface to avoid repeated calculations - the enemy ships could move in any direction and even jump! This is because the NavMesh library was designed for controlling human-like characters. It turned out that it was not straightforward to customise the internal behaviours of such Agents. There was also quite poor documentation on the library.

To achieve more realistic movement, a Finite State Machine like system was built over the NavMeshAgent layer. This also solved our interpretability problem. Now, the enemy AI could be in one of the following states:

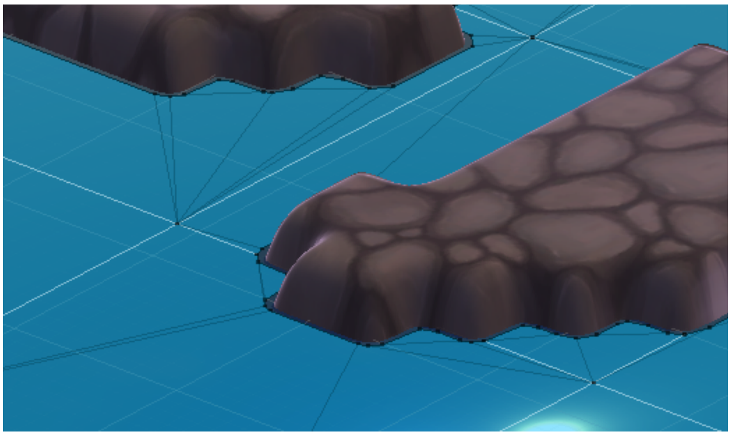
1. **Patrol**: moving around a set of points on the arena, searching for players to fight.
2. **Chase**: moving directly towards a player to fight them.
3. **Attack:** aiming and firing at a nearby player's broadside
4. **FightOrFlight:** deciding whether to flee from or attack a player who just damaged it.
5. **Flee:** moving in the opposite direction of the player attacking it

The current state of the FSM would determine what destination is given to the NavMeshAgent - however, the Agent cannot actually move its enemy ship. Instead, we disable its ability to change its own position, and just extract the path along the NavMeshSurface that it wants to take towards the destination. Another component, AIEnemy, interprets this path and moves the enemy in a ship-like manner accordingly.

In the end, this design was very successful. It was very easy to debug enemy AI during development as we could always trace bugs to specific states. Also, this AI system scaled well enough that we could support over 12 enemies at once with no performance issues.



Enemies using NavMeshAgent navigation paths



NavMeshSurface and some obstacles.

Enemy attacking a player

Designing and Creating our own models

We used Maya to create our own models which were then imported into Unity. We used a low polygon style of modelling to achieve higher in-game performance, as low frames per second can result in motion sickness when using a Virtual Reality headset.

The player ship is inspired by frigates from the 15th century. It has 3 masts for the sails, and a deck with different heights for the front, back and middle of the ship, where the cannons were positioned. One of the main concerns when modelling the player ship was to have a clear field of view for the player. With this in mind, the sails were positioned slightly higher so that the player could see everything around them without any obstruction. Furthermore, details such as railings, wood planks, stairs, and a base for the main mast were added to improve the aesthetics of the ship.

Model of the player ship.

The AI ship is inspired by cog ships. It has a single mast with a large sail and a flat deck with elevated railings. The cannons were positioned on elevated platforms at the front and back of the ship, clearly visible to the player. This was done so that players can easily identify these ships as a threat. Less detailing was added to these ships, as they are further away from players than their own ships, making fine details harder to appreciate.

Model of the AI ship.

Cannons were modelled in 2 separate pieces for the base and barrel of each cannon. By parenting the barrel to the base, cannons could animate where they were aiming by moving each part separately.

Model of a cannon.

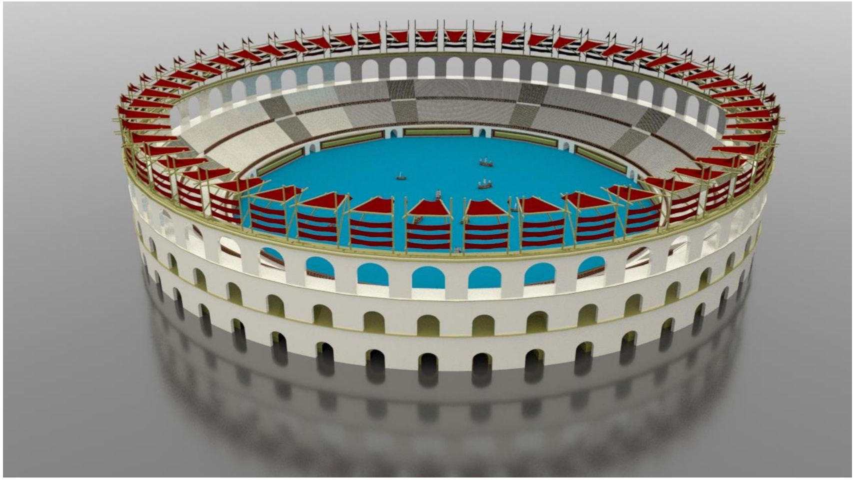
The sails for both ships were modelled using a dynamic modelling technique: nCloth for Maya. It works by applying physics to an object and simulating wind to deform the mesh into the desired shape. This technique creates realistic cloth-like meshes which are perfect for the sails.

To increase the aesthetic fidelity of the game, we also modelled different types of clutter that is scattered throughout the players' ships. These include barrels, crates, buckets, and cannonballs.

Player ship with cannons and clutter.

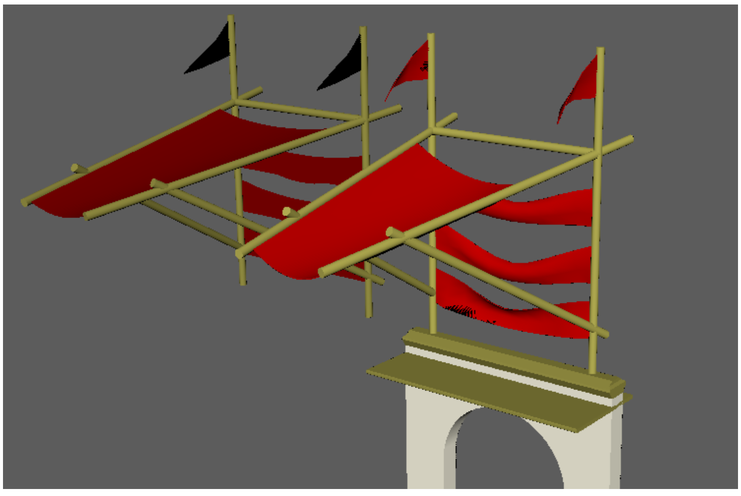
Colosseum: Naumachia is the art of naval battles that were staged in ancient Rome. So we needed a scene that with the first sight reminds you of Ancient Rome in its full glory. Since most of the colosseum images available online are of ruins we had to look at more places for inspiration. We drew inspiration from films/tv shows like 'Gladiator', 'Ancient Rome'. We also drew inspiration from colosseums shown in games like 'Assassin's Creed Origins & Odyssey'.

To model the colosseum we had to model individual mini-models like flags, poles, roof, windows, gates, railings, roman mouldings, seats, stairs. Then bring them together and use non-linear bend tools to create the arena.



Bird's eye view of Colosseum

This was a little tricky as everything needs to fit perfectly in place. Even if a single mini model was out of place by 1 mm, the model would come out broken after the merge and bend. You also needed to proportionate division to every mini-model or else they would all bend with different proportions. The cloth design of the flags were generated using nCloth dynamics along with simulating respective air and gravity fields.



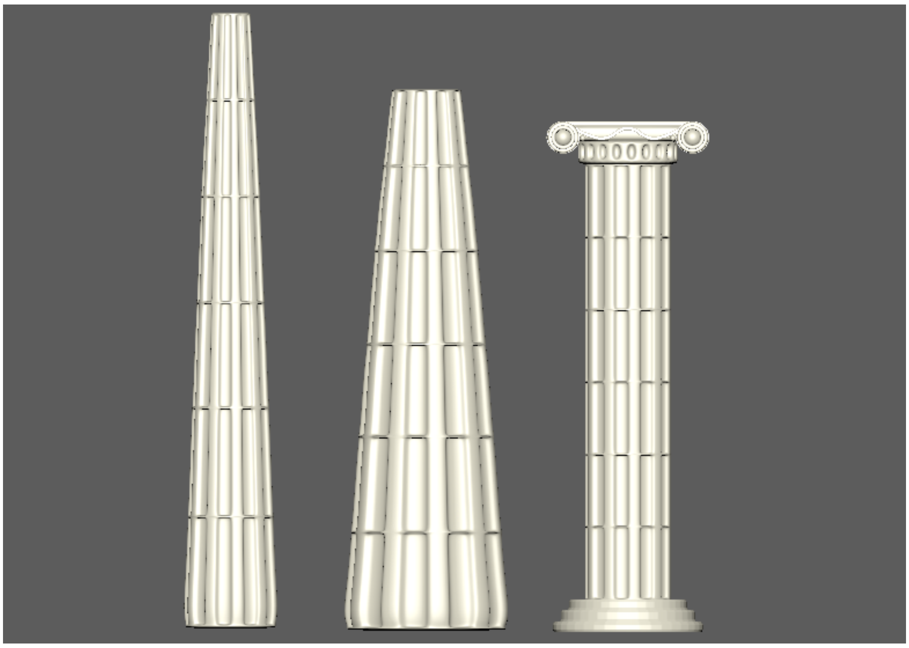
Creating realistic design for the flags of Colosseum

The size of the gates of the colosseum needed to be exactly proportionate to the ships. If the gates were much smaller than the ships, upon increasing the size of the colosseum, everything else in it would become very big in proportion and would dwarf the ships. The opposite would happen if they were too big, i.e. the colosseum would appear too small for the ships. The colosseum had to be modelled in a way to make everything perfectly proportionate in size.

Volcano Terrain: We needed an attraction in the centre of the colosseum. So we decided to have a volcano which would give the game an arcade look. With some research acquired skills and video tutorials we were able to create a volcano terrain in Maya. It was modelled with the tools like Z-brush, contour mapping, bump maps, etc. The volcano terrain has deep ridges and depths to allow space for the lava river to flow from the volcano.

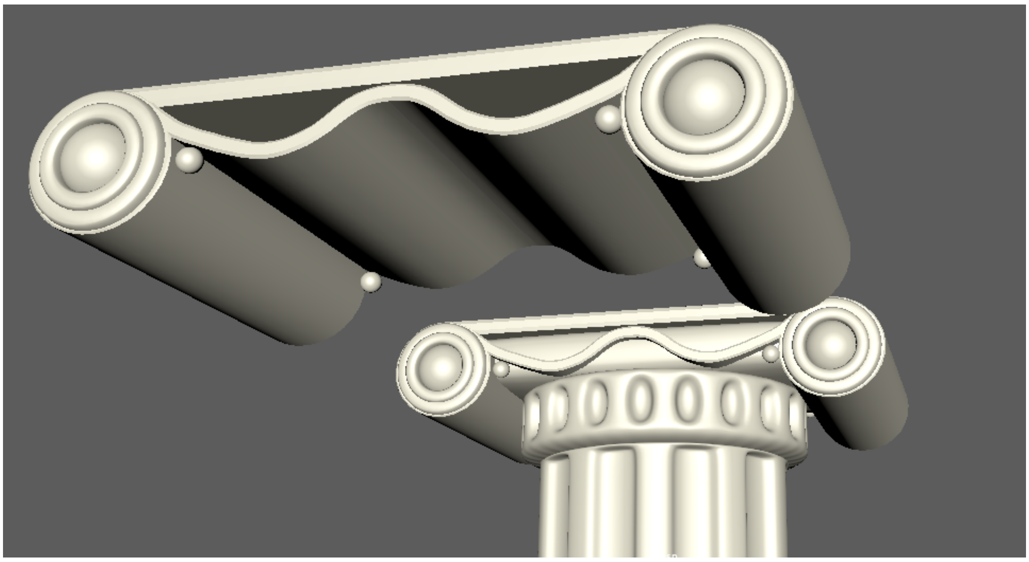
Volcano

Greek Columns and Arches: To improve the gameplay we needed to design obstructions inside the colosseum and make it more interesting for the players to navigate around. These were created with the help of deform, bend and curve tools.



3 types of Columns designed

We made sure that each of these structures strongly depicted the Roman architecture to stay in tune with the theme of our game. The trickiest bit was making a wave-like roman moulds on the top of the pillars. The curve tool was used to create this mould with just cube polygons.



Sculpted moulds on the top of Columns

The arches were made for the ships to pass through them to receive a special feature in gameplay. The arches were made with the use of the bend tool, multi cut tool and target wield tools.

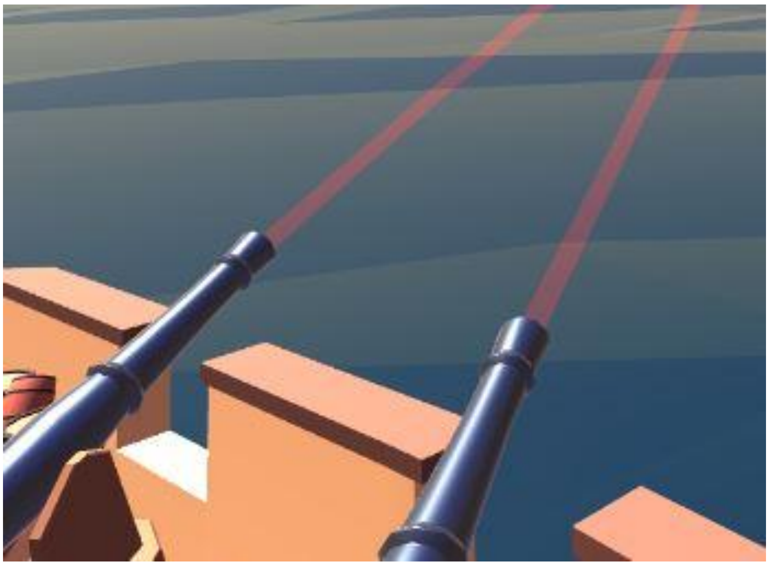
Greek style Arches

Firing System

The firing system started as a basic aiming and firing using mouse controls and eventually evolved into using VR input, such as head tracking to control the aiming of the cannon balls and voice commands to initiate firing sequences.

One of the main technical challenges was developing realistic physics principles for the movement of the cannon balls themselves. This became a primary objective after feedback from our initial alpha review as it would help develop immersion into the game.
To do this we utilised parabola equations, and Unity's raycasting features, to predict the end point of the cannon ball's trajectory. This allowed accurate modelling of trajectory and realistic physics which helped solidify cannon firing in relation to the movement of the player ship. This was improved when we added rigidbodys to the ships to allow the impact of forces on their movement.

This also allowed the cannons to utilise Unity's lookAt function so as to swivel to the point where the player is aiming. This enabled our game to feel responsive, with added object animation and immersion.



Old iteration of aiming arcs using the deprecated line renderer.

To visualise the firing arcs we initially attempted to use the in-built unity line rendering scene object. However, we quickly realised this was not an ideal solution due to the functionality being deprecated. We therefore created our own bespoke solution which involved utilising a trajectory model and splitting it up into individual coloured line segments. The overall effect was that of a dotted line which was both more user friendly (easier to see where you were aiming) and more in keeping with our game's cartoonish aesthetic than the line renderer.