

# Insane Ian

INSANE  
IAN



## Team A

Ye Olde Pubby McDrunkface

Alex Welsh

Jordan Taylor

Amber Rothera

Jed Priest

Matthew Crees

George Sains

**Team Manager**

**Lead Programmer**

**Lead Designer**



University of  
BRISTOL

## Contributions

---

**Alex Welsh**

*Weighting: 1*

A handwritten signature in black ink that reads "A welsh". The "A" is large and stylized, and "welsh" is written in a cursive, lowercase style.

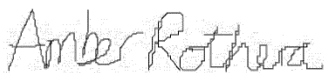
**Jordan Taylor**

*Weighting: 1*

A handwritten signature in black ink that reads "jtaylor". The letters are connected in a cursive script.

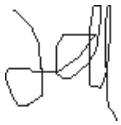
**Amber Rothera**

*Weighting: 1*

A handwritten signature in black ink that reads "Amber Rothera". The signature is written in a clear, slightly cursive style.

**Jed Priest**

*Weighting: 1*

A handwritten signature in black ink that reads "Jed". The letters are stylized and connected.

**Matthew Crees**

*Weighting: 1*

A handwritten signature in black ink that reads "Matthew Crees". The signature is written in a clear, slightly cursive style.

**George Sains**

*Weighting: 1*

A handwritten signature in black ink that reads "G Sains". The "G" is large and stylized, and "Sains" is written in a cursive, lowercase style.

## Top Five Contributions

---

- 1. We replaced Unity's physics components with our own, using C++ to directly access Unity's internal physics engine, enabling deeper access to simulation data and finer control over physics events.*
- 2. We implemented real time networked physics and dynamic interactions with lag compensation algorithms and smooth correction of vehicle velocity and rotational velocity across up to 12 clients.*
- 3. We created a detailed soft body physics solution by developing scripts for intricate, multi frame mesh deformations.*
- 4. We Implemented high quality audio communication across multiple users and integrated the feature seamlessly into our gameplay.*
- 5. We utilised a meticulous team process with over 100 team meetings and 1200 content commits to our version control repository (>1800 including merges) to tackle the ~260 tracked tasks we created.*

***Demonstration Video: The Everything of Insane Ian***

<https://www.youtube.com/watch?v=iUHVXT2AJXs>

## 1. Team process

- Held consistent meetings 3 days a week, often twice a day, both in the morning and afternoon (see Team Process and Project Planning section: page 9)
- Assigned, tagged, and tracked over 260 tasks using the Microsoft Teams [1] (see Team Process and Project Planning section: page 9)
- Created a detailed flowchart during initial stages when building baseline systems that were deeply interconnected (see Team Process and Project Planning section: page 8)

## 2. Technical Understanding

- Used concurrent resource management and error correcting principles to ensure consistent game state. (see Technical Content section: page 20)
- In depth research on the internals of the game engine and the build process. (see Technical Content section: page 25)
- Researched and developed efficient algorithms to simulate the effects of collisions on visual meshes. (see Technical Content section: page 29)
- Used efficient transmission protocols to synchronise vehicle physics. (see Technical Content section: page 20)
- Used multiple latency compensation algorithms to produce enjoyable weapon behaviour. (see Technical Content section: page 22)

## 3. Flagship Technologies Delivered

- Implemented efficient, multi-frame mesh deformations (see live demo, video at 5:22, Technical Content section: page 29)
- Delivered networked vehicles and other physics objects, with seamless velocity and rotational corrections (see live demo, video at 1:27, Technical Content section: page 20)
- Created networked projectile and hitscan weapons (see video at 4:04 and Technical Content section: 22)
- Built a high-quality voice communication system, reactive to game environment. (see live demo, Technical Content section: page 23)

## 4. Implementation and Software

- Replaced Unity's physics components with our own by directly accessing the underlying physics engine C++ code using C#/C++ interoperation. (see Technical Content section: page 25)
- Created a polymorphic weapon type interface to allow easy prototyping and integration of various weapon systems. (see Technical Content section: page 21)
- Created AI agents to fill out teams to ensure playability with any number of people. (see Technical Content section: page 22)
- Created a voice chat that is compatible with WebGL Unity games using C#/JavaScript interoperation. (see Technical Content section: page 23)
- Created a dynamic cinematic camera system to showcase our soft body physics. (see Technical Content section: page 23)

## 5. Tools, Development and Testing

- Utilised agile-like week-long sprints with consistent team tests on Friday afternoons. (see Team Process and Project Planning section: page 9)
- Managed a >1800 commit GitHub repository with over 50 branches. (see Team Process and Project Planning section: page 10)
- Play-tested our game on over 20 people aged between 17 and 54 (see Software, Tools, and Development, page 19)

- Unit tested our core game data management system. (see Software, Tools, and Development, page 19)
- Profiled all our game code to find and remove inefficiencies. (see Technical Content section: page 29)

## 6. Game Playability

- Controls have been kept very simple to improve accessibility and are described in a clear tutorial, as well as screen prompts help the user in game (see demo, Abstract: page 7).
- Developed player convenience features directly from observed gameplay, including orientation correction and auto-ram. (see Team Process and Project Planning section: page 20)
- Reconfigured gameplay based on feedback from testing to enhance approachability (see Team Process and Project Planning section: page 20)

## 7. Look and Feel

- Combined both aesthetics and performance in a low polygon design. (see Team Process and Project Planning section: page 18)
- Hand made every single asset in the game except for use of free typefaces, our skybox, and the engine noises [2][3][4][5]. (see Team Process and Project Planning section: page 9)
- Created an AI camera to enhance exciting moments with adaptive camera angles and slow-motion, particularly highlighting deformations. (see video at 7:25, Technical Content section: page 23)
- Created a custom shader to give our game a unique aesthetic. (see Technical Content section: page 28)
- Created 40+ original models in Autodesk Maya including vehicle models. (see Individual Contributions, page 13,15,16)

## 8. Novelty and Uniqueness

- Soft body mesh deformations from networked collisions, running within a browser. (see live demo, video at 5:22, Technical Content section: page 27)
- Hand made every single asset in the game except for use of free typefaces, our skybox, and the engine noises [2][3][4][5]. (see Team Process and Project Planning section: page 9)
- Constructed a unique chaotic, playful atmosphere enhanced by both visuals and gameplay. (see live demo, abstract: page 6)
- Created a cooperative vehicle control system for two players and gameplay which encourages communication and teamwork (see live demo, abstract: page 6)

## 9. Report and Documentation

- Clearly documented our gamestate tracker (see supporting videos: [https://youtube.com/playlist?list=PL5tU7\\_CTItxk5dTwjiSiRlXF52ZlZa31N](https://youtube.com/playlist?list=PL5tU7_CTItxk5dTwjiSiRlXF52ZlZa31N) )
- Clearly documented our weapons systems (see supporting videos: [https://youtube.com/playlist?list=PL5tU7\\_CTItxk5dTwjiSiRlXF52ZlZa31N](https://youtube.com/playlist?list=PL5tU7_CTItxk5dTwjiSiRlXF52ZlZa31N) )
- Clearly documented our networking optimisations. (see supporting videos: [https://youtube.com/playlist?list=PL5tU7\\_CTItxk5dTwjiSiRlXF52ZlZa31N](https://youtube.com/playlist?list=PL5tU7_CTItxk5dTwjiSiRlXF52ZlZa31N) )
- Clearly documented our vehicle networking systems. (see supporting videos: [https://youtube.com/playlist?list=PL5tU7\\_CTItxk5dTwjiSiRlXF52ZlZa31N](https://youtube.com/playlist?list=PL5tU7_CTItxk5dTwjiSiRlXF52ZlZa31N) )

# Abstract



Figure 1: The introductory loading screen

## Overview

Insane Ian is a multiplayer action racing game in which teams of two cooperatively pilot vehicles and attempt to score the most points within a three-minute match against other teams. Points are scored by destroying the vehicle of another team or by holding the Gubbinz; an object which powers up your vehicle but makes you much more of a target. We have made extreme optimisations to allow networked mesh-deforming collisions to run in a browser.

## Story and setting

The game takes place in a futuristic wasteland world. Technology was advanced before most of humanity was wiped out, and all that is left are scraps of what once was. The characters in our game are scavengers who have created weird and wonderful vehicles and weapons from these scraps and are always on the lookout for upgrades. The players fight over resources to drive and survive.

The gameplay environment is a desert scene with an abandoned town, cave system and desert terrain. After a considerable amount of testing, we settled on a central arena with 3 different paths leading off and looping back in. Each path has its own theme. We found this setup allowed us to blend different design concepts and path topographies for unique gameplay in each area. Scattered throughout the entire map are interactive objects which shatter in a collision. This does add to the already heavy physics computation, so optimisation here was key. These objects bring add an extra layer of depth of interaction between the player and the world we have created.

## Roles

Players can play as one of two roles: driver or gunner. Drivers have control over the motion of the vehicle and have both offensive and defensive responsibilities. They must keep themselves out of harm's way but can also deal considerable damage to opponent vehicles by ramming into them. Gunners on the other hand have a clear offensive role. Their objective is to deal as much damage to enemies as possible. Communication between drivers and gunners is essential for maximum success. We have built in several features which specifically reward teamwork. For example, gunners can select a target vehicle for the driver. If the driver then collides with the target, increased ramming damage will be dealt.

## Gameplay

From the opening menu of the game, players can choose to either create a lobby or join one created by someone else. Players can select a role within a given number of teams. Any slots left empty by the time the match starts will be filled by AI 'bot' characters. These bots can either fill in one empty slot on a team with a player or can make up teams of their own if the host chooses. Within the lobby screen, all players can choose to join a voice call and talk to everyone else in the lobby. Once the host has chosen a map, they can navigate everyone to the garage screen, where the driver of each team can select their vehicle. When a gunner player is on a team with an AI driver, the player can select the vehicle. There are 3 choices of vehicle for players, each with different weapons, health values and driving characteristics. Upon entering this screen, teams are locked into separate voice channels to only communicate with their partner. The game can then proceed to a match.



Figure 2: An in-game screenshot of the beginning of a match



Vehicles spawn in a ring around the Gubbinz and are released after a short countdown. Teams then proceed to fight for points, gained by destroying enemy vehicles or holding onto the Gubbinz. When in possession of the Gubbinz, players also will receive health regeneration and a slow charge up of ultimate abilities. Ultimate abilities are very powerful single use abilities for both the driver and the gunner. The driver's ability is a powerful nitro boost. The gunner's ability is a more potent weapon with limited ammunition, which is unique to each vehicle type. Each player's ability is earned by their teammate. We wanted this feature as it adds motivation for communication and coordination between players in a team. The driver earns the gunner's ultimate by finding pickups scattered throughout the world. The gunner earns the driver's ultimate by dealing damage to opponents. While holding the Gubbinz both abilities are slowly charged up. UI elements on screen show important statistics for both driver and gunner. These include current vehicle health, current ultimate charge level and match time remaining. Gunners see their vehicle's remaining ammunition, which regenerates over time.

One of our key technologies is soft body physics. During gameplay, this is featured when vehicles deform during heavy collisions. We replaced our game engine's physics components to have smooth, multi-frame mesh deformations. To highlight this, collisions deal considerable damage and will instantly give you the Gubbinz if your collision destroys the opponent with it. To improve accessibility, an auto-ram feature has been added which will direct the vehicle towards an opponent with a strong force.

Once the time limit has been reached, the players are transported to an end screen displaying the final scoreboard as well as a podium scene. Voice channels are opened and everyone in the lobby will again be able to speak to each other again.

### Controls

The controls of the game differ for each role: driver, or gunner, and we have kept the controls for both roles simple.

The driver uses the W, A, S and D keys on a keyboard to drive and steer the vehicle. Both roles use the Q key to

activate their ultimate abilities. The auto ram feature is activated with the E key. Whenever use of the Q or E keys is recommended, visual prompts will be shown to the driver, to improve ease of use of the game.



Figure 3: Screenshot of the driver controls tutorial

The gunner uses their mouse to aim at opponents and uses left click to fire their weapon. They can use their scroll wheel and space bar to zoom in and out as desired. The gunner also uses the Q key to activate their ultimate weapon. Another feature we added which further encourages teamwork is the gunner's ability to mark target cars with right click. The driver can see the marked target.

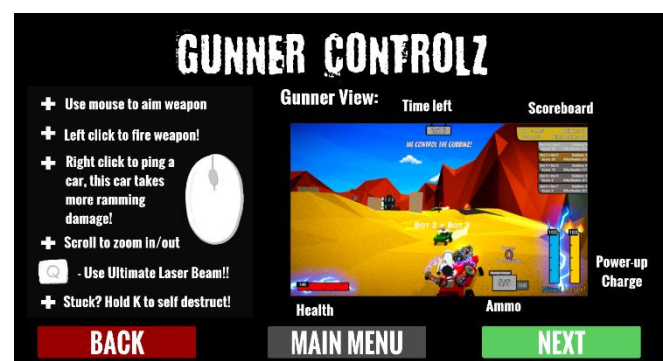


Figure 4 : Screenshot of the gunner controls tutorial

We also added an option to allow drivers and gunners to self-destruct their cars by holding down the K key for 3 seconds in case they get stuck. However, we have designed the map and cars, as well as how collisions occur to greatly reduce the chance of getting stuck in the environment. In our playtests, not one person needed to use this self-destruct function.

# The Team Process and Project Planning

## Overview

Our team process and project planning has had two distinct stages: before the MVP and after the MVP. In the beginning the clear and simple goal was to build a baseline project for our MVP deadline. At such an early stage, anything we built would be strongly connected to other parts of the project. We also wanted some way of visualising the order in which intertwined tasks needed to be tackled. The solution for us was to create a flowchart, mapping out our progress from the most basic elements to a working MVP. Once this was complete, we moved on to an agile style week-long sprint process, which strongly fit our requirements.

Our task management and meeting structure has remained constant throughout. We looked at multiple issue tracking software products but settled on Task by 'Planner and To Do' [1]. The simple reason for this is that it offered all the functionality we needed whilst also being integrated into our established Teams channel. Said channel is also the site for all our team meetings. The format of these has been such that every Monday, Wednesday, and Friday since the start of the project, we have some sort of meeting. There have been only two days since the 1<sup>st</sup> of February that we have not held a scheduled meeting, these are the 5<sup>th</sup> and the 7<sup>th</sup> of April during Easter.

## MVP Flowchart

The project started as many little individual programs which each focused on a single challenge, for example vehicle controls or networking. We quickly found that some of these would require a lot more attention than others and would need varying levels of work to reach a functioning stage. It was also unclear when these threads would need to be merged and what order we should best tackle them. We therefore decided a flowchart of tasks was the best place to begin. We chose Lucidchart to create the diagram for two reasons: first, it has excellent interoperability with Teams, allowing us to switch to it very quickly in a meeting; second, it has a compelling feature set including not just the standard array of shapes and tools but also allowing multiple users to view and edit in real time [6]. Figure 5 shows the completed MVP chart. We colour coded the chart to make quick referencing simple with white being "not started", amber being "in

progress", yellow being "requires testing" and green being "complete".

The flowchart significantly improved our ability to prioritise tasks. We quickly realised that setting up the networking was key to many other things, so devoted a lot of time early on to try and get it right first time. We could also now see what tasks were available to work on without any precursor requirements. Finally, it also gave a good indication of how much we had accomplished and how much work remained. We used this information to both side-line features when we were behind and to write in extras when we were ahead.

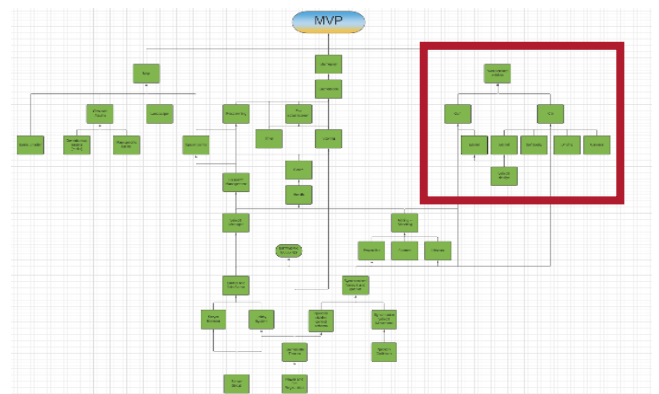


Figure 5: The completed MVP chart, with Figure 6 highlighted

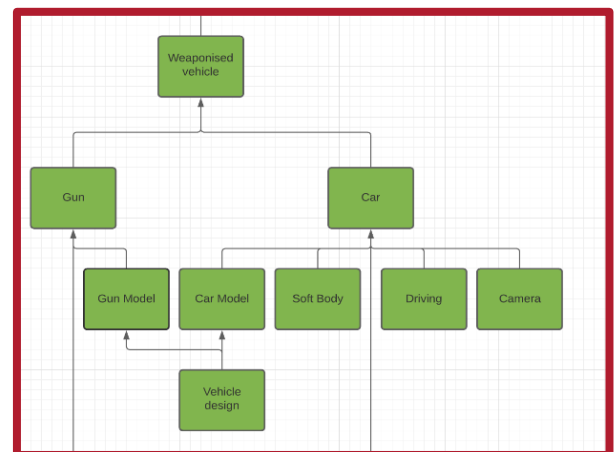


Figure 6: A close-up of the vehicle section in the complete MVP chart

Overall, we fully believe it was the correct decision to create this flowchart, and that it was the correct decision to not pursue any subsequent duplicates. The nature of our workflow changed significantly after our MVP. Once the base of our game was built, we could expand with features that were no longer as tightly interwoven.



Development of new elements became much easier, so we are glad we only created the single flowchart.

## Weekly meetings

Our weekly schedule had meetings every Monday, Wednesday, and Friday, at either 10am, 2pm or both. We chose those days as it meant every Monday, we could discuss plans for the week, Wednesdays could be used to discuss challenges or ideas, and Fridays were reserved for integration, testing and reflection. Each team member has been participating in their own mini project so, while each person's time is their own, it was a common decision to devote Tuesday and Thursday to that. Each week was often devoted to implementing a single feature per person, so Fridays naturally became a day to integrate work from different team members and attempt to playtest the changes together. We cannot definitively say we used an agile process, as we were too inconsistent, but we absolutely took inspiration from week-long sprints. One challenge we faced was that we often would not be able to play the game for several weeks. This is due to our ambitious key technologies and the complex debugging issues they present. An example of this was a physics bug we found after integrating our own physics components. We were blocked from true gameplay testing for nearly three weeks.

Our reasoning for two meetings a day changed throughout the project. Initially we only had morning meetings, but we found that often teammates would create a feature in the morning then want to discuss, test, or bugfix in the afternoon with help. We found ourselves repeatedly making 2pm meetings so they soon became more formal. With a second meeting on the books each day, we made the decision to try and keep one as mandatory – getting every member in the same call – and one as optional so anyone could bring up points to the team manager or seek help fixing bugs.

We have remained extremely consistent across the project. The 5<sup>th</sup> and 7<sup>th</sup> of April are the only days since the beginning of February this year that have not hosted a normally scheduled meeting. This is simply because we decided to reduce workloads over the easter break.

If we were to start the project again, we would have to look closely at overworking and be careful to balance time on both this project and our other commitments. Some members of the group focused too much on the game for the first two months of the project, and subsequently needed to divert significant amounts of time to their

personal mini project in the later stages. Having a clear distinction from the beginning of when to work on what may have been a better approach.

## Issue tracking

From our first meeting, we have been tracking our tasks via our Teams channel. Every task we have made is designated a single bucket and assigned at least one tag. Buckets relate to the subject matter and tags relate to the type of action to take. For example, creating a UI element would be classified in the UI bucket and tagged with “new feature” and “art & design”. The issue tracker has recorded over 260 tasks throughout the project, Figure 8 is a screenshot of the tracker mid-way through the project.

Having the tracker gives team members an excellent insight into who is working on what at any given time. One of the most common uses of the tracker has been to store ideas for the future. With the ability to look for unassigned tasks, we have streamlined the delegation process significantly.

After using the tracker for several months, the only grievance we have is that tasks cannot be linked to each other; for example, as a precursor. On reflection, this is a small price to pay for the added convenience of having our tasks, meetings, and file storage all in one place. The one solution to offer both advanced features and have Teams' integration is Microsoft's Project software, however the software is proprietary, and we have no way of acquiring licences for free.

## Repository management & File sharing

We have used GitHub to host a remote repository of our source code [7]. To maintain a distinction between different areas of the project, we have used branches liberally. As of writing, we have used more than 50 to subdivide our work. We set out clear guidelines from the beginning to have one person work on a branch at a time, and to ensure the build is stable before merging in changes. It has been policy to commit and push often, which has saved us several times from data loss. We have made more than 1200 content commits to the repository, and more than 600 merges. Commit rates have been relatively even across the board, with the 4 main software developers of the team making 200-300 content commits each, and the two designers of the group making the rest.

This is expected as it does not take into consideration additional work in programs such as Autodesk Maya [8].

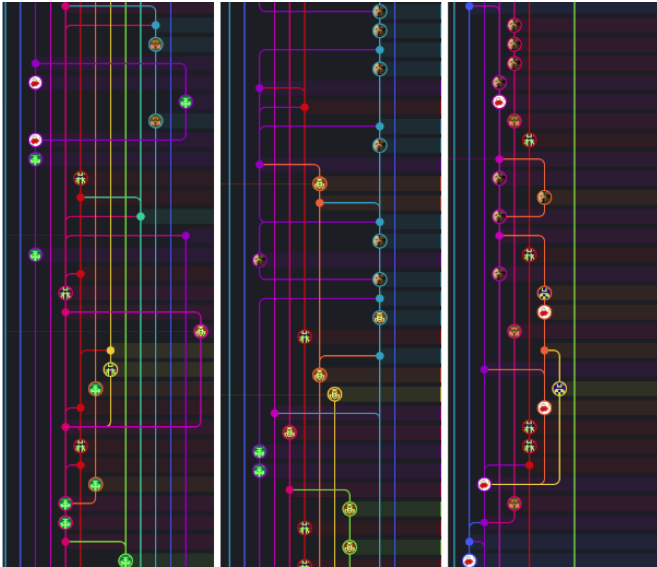


Figure 7: A selection of snapshots from our repository branches, visualised in GitKraken [14]

We have taken data security very seriously throughout the project. For most low-risk files like meeting notes, we have opted to use the SharePoint page associated with our Teams channel. This is simply for convenience. Being a closed team, only university staff and students can gain access and view or make changes to files. For more secretive data, we have shared access only between team

members. Our repository has been kept private from the start, to avoid spying or vandalism. Similarly, our Google Cloud project has restricted access except where necessary for internet access. This report was developed in a private OneDrive folder, and access was manually granted to team members only.

## Crisis management

Over the course of our project, every member of the team had at least one major personal issue occur, resulting in a break from work.

From the beginning, we all decided on an open policy of trust. We have never questioned any person’s motives for missing a meeting or not being able to deliver a feature. We have made it clear that there is absolutely nothing wrong with taking a day off. Mental and physical wellbeing must be the highest priority of every member in the team. Each teammate is trusted to make judgments about their own situation and to put in a proportional amount of work. We have worked to minimise downtime when a team member is unavailable. A large amount of general development and bug fixing is done in team-wide meetings by sharing screens. A core reason for doing this is so multiple team members know how each section works. This additional redundancy of knowledge is exceptionally useful. It both allows for more effective troubleshooting, and for a better knowledge base with a subset of the team.



Figure 8: A screenshot of the Microsoft Teams task screen from the 15th of March [1]

# Individual Contributions

---

## Alex Welsh – Team Manager

*Focus: Administration, Vehicle control dynamics, Web hosting*

### **Team Management**

As team manager, I have taken on the administration duties of the group. Near the beginning I set the precedent of regular meetings to ensure every member of the team has the most up to date knowledge of what everyone else is doing. I have also been the initial point of contact for other parties such as university staff and composers. Keeping the team on track has been a major part of my role. I have put a lot of work into creating and assigning tasks in our Teams tracker as well as following up with teammates to ensure we work as efficiently as possible. Fortunately, this has mostly been reminding team members to take breaks and not overwork themselves. This has in total taken an average of 4-5 hours per week.

### **Vehicle control dynamics**

One of my largest contributions to the game has been the vehicle control systems. These include simple features such as mapping inputs to wheel torques, and more complex additions such as different tyre friction models for different surfaces throughout the map. I initially researched other implementations of driving physics in games such as Forza Horizon 4 and took inspiration from real world components like anti roll bars. I worked with Jordan to ensure the object-oriented control code would easily be interactable for AI bots as well as humans. Between initial creation and maintenance throughout the project, I have spent around 4 weeks on vehicles.

### **Web Hosting and builds**

I have been the main manager of the Google Cloud hosting of our project [9]. Initially I created testing buckets which provide the ability to drop-in game files and have a playable web build in seconds. Since then, I acquired insaneianguame.com and linked it up to a load balancer and bucket for high quality live builds. Related to this is my work on build settings and setup. I have been

the primary researcher on compiling builds and tweaking settings to maximise performance. I led the change from asm.js to WebAssembly which brought considerable efficiency improvements. Web hosting has taken approximately 2 weeks.

### **Particle effects and QOL features**

I have been the architect of many small features that contribute to quality of life and the feel of the game. I created a system for dust and gravel trails behind vehicle tyres, based on the surface they are driving on. Similar to this is a visual damage system that indicates a vehicle's health by emitting varying levels of smoke. I also added various small systems such as those to keep players upright and make them flip randomly during destruction animations. Early in the project I adapted a free skid mark system asset to dynamically place decals on the terrain. This was later cut out both because it was not created fully by our team and for performance reasons. This has overall taken approximately 2.5 weeks.

### **Gameplay support**

Alongside my normal duties, I have offered programming support for some gameplay features. During development, the objective of the game was going to be a race between checkpoints. I wrote the scripts to allow this and prototyped adjustments as they came up. I have spent about 1.5 weeks on this.

### **Report Writing**

I have taken responsibility of managing the creation of this report. I contributed heavily to the Nine Aspects; Team Process and Project Planning; and Software, Tools and Development segments. I have also reviewed personal contributions from all other team members in the Individual Contributions section and Technical content. This has taken a week in total.

# Jordan Taylor – Lead Programmer

*Focus: Feature implementation, Networking, VFX & SFX*

## **Lead Programming**

My responsibility as the team's lead programmer has been to develop Insane Ian's key technologies and integrate the technologies of other team members into an ultimately a playable game. I have created the overall framework for a cohesive development environment. Seamless Integration of technologies has consumed most of my time on this project.

## **Networking**

The first technical task of the project; I had the responsibility of rapidly setting up the backbone of the game's multiplayer using Photon Unity Networking (PUN). A playable prototype of the game was operational within a fortnight. My role in networking continued throughout development as new features were introduced – pioneering an 'authoritative master' network model with client-side concessions.

## **Weapons**

I was responsible for creating the mountable weapon system framework used by gunners in Insane Ian. The system was created with ease of use in mind, using a layer of several interfaces and weapons of polymorphic type to allow any type of weapon to be mounted to any type of turret. I also had responsibility of writing all weapon code, weapon behaviours, particle effects, animations, and their lag compensation/corrections algorithms. Weapons took roughly three weeks of work time.

## **AI Bots**

One of my key contributions was the creation of bots, which capitalised on the design patterns and standards I had developed alongside the team. Bots rely on the same driver and gunner interfaces as players, in addition to being controlled via the authoritative-master model. Bots were created over the course of a week and continually supported throughout development, taking a sum of two weeks.

## **Smart-Camera**

An essential contribution to the game, I was responsible for creating a driver camera that would show off our car crashes and collisions in all their detail. Measurements such as estimated time to crash, collision impulses, and hit directions all factor into the camera's placement algorithm. The smart camera was continually tested and tweaked, taking several weeks of work to perfect.

## **Visual FX and SFX**

When not attending to programming duties, I was responsible for creating most sound effects and particle effects seen in Insane Ian. These included weapon fire and impact sounds, collision effects, car turbo noises, as well as their particle effect counterparts. These were created using a suite of external tools including Audacity and several synth pads.

## **Video**

I was responsible for creating the technical overview video for the project submission. This included writing the script, editing the video, and recording technical demonstrations over the course of a week.

## **System Interactions**

Perhaps the most important task I was charged with as lead programmer was to ensure that all game systems people had developed were able to interact reliably with each-other. This was often over a network with many potential race conditions. For instance, upon the spawning of a player, driver and gunner controls must be divided between appropriate players and bots. This relies heavily on the interactions between the master client, the *gamestate tracker*, all clients, and multiple instances of vehicles as all network view ownerships are dynamically allocated.

## **Telecaster**

I made the telecaster; however, this feature was cut from our final version due to performance constraints imposed by WebGL. This took a week and a half.

## Amber Rothera – Lead Designer

*Focus: Overall design of the visuals. Vehicle, asset and map modelling*

### Overall Design

My primary goal as lead designer is to create a world which both shows off our hard work on back-end technologies and tells a story to give players the best experience possible. I believe it is essential for the whole team to be involved in the design process, so everyone has a chance to share new ideas and opinions. The gameplay of *Insane Ian* is intense and aggressive by nature. Therefore, to keep our game light-hearted and fun, I decided to make the visuals and atmosphere playful, choosing bright colours and a low poly modelling style.

### Vehicle Design

I was responsible for designing the initial prototype car. To do so, I worked closely with the rest of the team to ensure that my model fitted an extensive list of back-end technical requirements; for example, I ensured the body of the car had large flat surfaces that would show off the soft body deformations. I then modelled a vehicle I designed called *The Interceptor* as well as Jordan's design, *The Spitfire*. Each vehicle took 3 days to initially model all the components in Autodesk Maya, before I rigged them in Unity. This was an iterative process, and all the vehicle models were continuously altered and adjusted through the project. Overall, vehicle design and creation took approximately 2 weeks of time.

### Map Design

I was the main designer of the maps we have used throughout the game. My aim was to create a world that is exciting and engaging, while reinforcing the main gameplay. I created the first 'Crater' map to test different gameplay modes and how players would interact with each other and the environment. This initially took 3 days to model. We later changed the gameplay, so I designed a new map to fit it called *Crash Valley*. The map consists of three looping tracks surrounding a central crater. I modelled the central crater and two of the looping sections: the town, and the desert. This took me 5 days to model. I also created the background environment for the menu, lobby, and garage scene. Overall, I spent 2 weeks designing and creating maps.



Figure 9: An array of assets I designed and modelled

### Map Assets

While developing the maps, I created many assets to fill the world with to enhance the gameplay experience. Some of my assets can be interacted with by the players, such as destructible objects like crates, cacti, shacks and exploding barrels. In total I made over 20 different map assets of varying complexities, from simple cacti to complicated buildings. Each took between an hour and a day to create and program in, totalling around 2 weeks of work.

### Menu User Interface Design

After Matt created the initial menus and lobby areas, I then adapted them to give a consistent look and feel to our game. I researched fonts and colours to use across the interface that would both fit the tone and be easily read and understood. I also created a clear and concise tutorial which explains the objective and controls of the game, using images and sprites. Finally, I also designed and created multiple logos for the game, as well as several sprites used in the game. This took a week in total.

### Music Management

I worked closely over several months with our 2 composers, Matthew Collins and Vivienne Youel, who created 6 tracks for our game. I have had bi-weekly meetings with them alongside regular email contact. I integrated each piece into the game along with code which let them transition smoothly. This took 5 days of time in total.



## Jed Priest

*Focus: Key technologies, Physics implementation, Networking*

### **Soft Body Deformations**

I worked with Matt on the first iteration of soft body deformations. We created a system that allowed explosions and collisions to happen at any point on the body of the vehicle and would dissipate the force of collisions and explosions in a reasonably realistic manner. This first iteration took a week. I then spent an additional week further developing this before deciding that Unity's in-built physics engine does not offer enough flexibility for our needs.

For the second iteration, I used PhysX's internal simulations to draw out the collisions to be more visually impressive. I then modified the code from the first iteration to account for the continuous, rather than instant, nature of the collision. This also took a week.

### **Custom Wrapper for PhysX**

The instantaneous nature of physics calculations in Unity meant that collisions were disappointing, and our soft body deformations were not being used to their full potential. I decided it was necessary to access the physics engine directly and make changes allowing collisions to occur over multiple frames.

I first built an external version of PhysX for testing in the editor and then set up the web build to compile with extra C++ to access the physics engine in the browser. This took two weeks.

I then built a wrapper around it, making drop-in replacement components for Unity so that the transition to the new PhysX system was as easy as possible. This took a week.

While integrating the new PhysX system, I supervised this process to ensure that everyone knew how to use the new system and that everything was switched across correctly. I spent a significant amount of time fixing bugs that were uncovered during the transition as well. This took at least two weeks including long term support.

### **Gamestate Tracker Refactor**

To improve the stability of the game, there came a point where we had to replace a large portion of the networking system, which had previously been built incrementally with various additions and quick fixes, with something more reliable. To do this I replaced the gamestate tracker, which was originally a collection of variables with remotely callable getters and setters, with a simple database. I designed it to be lightweight so that it had minimal impact on game performance. I also made sure that it was resistant to data races and collisions, so that we would not have any errors which could adversely affect player experience. This took a week and a half.

### **Optimisations**

As there were no guarantees that the players would have powerful computers, we had to make sure the game would run well on low-end machines. To do this I ran Unity's inbuilt profiler on our code to highlight areas that were particularly inefficient in terms of speed and memory usage. I then modified them to give better performance and make sure they would run well on our target devices. This took about a week total over the course of the project.



Figure 10: Deformations caused by collisions in-game



# Matthew Crees

*Focus: User interface, User experience, Voice communication*

## **Voice Chat**

I have hosted our own instance of a PeerJS server as a Google Cloud App Engine to handle peer-to-peer voice communication. For this to work with players behind symmetric NATs (e.g., routers), I also set up a Coturn instance on a Google Cloud Ubuntu VM to route a connection between players via TURN/STUN servers. I wrote a custom WebGL template to be used when we build our game that includes the JavaScript code which handles setting up and connecting Peers, and muting/unmuting audio streams. This is then controlled from C# scripts through an intermediary JavaScript library, all of which I also wrote. This took about a month.

## **In-Game UI**

I created the in-game UI in its entirety. I made the artwork myself, with a focus on showing all the information that a player may need clearly and concisely. I used a combination of time-limited animations and a concise layout to make sure that the UI does not feel cluttered or obscure the gameplay. I also programmed the functionality of the in-game UI. This includes the scoreboard, health and ultimate bars, timer, and the post-match scene. The scoreboard, timer, and final scene have extra complexity as I had to tie them in with our networking solutions. I had to update this UI multiple times throughout the project to reflect changes and new features in gameplay mechanics. I also created an in-game tutorial system that would be dynamically displayed based on which stage of the gameplay players were in. However, this was eventually replaced by a more streamlined menu-based tutorial. In total, this all took about a month.

## **Menu UI**

The menu was the first thing that we had in our project, created by Jordan. I overhauled his menus with a focus on user experience. I created custom artwork and

animations that are designed to be clear and scalable with screen resolution and designed and implemented additional menu screens to let the player navigate easily. I also created a new server browser, which scales to have multiple pages if enough games are created. Additionally, I altered the lobby screens to hide buttons and information from players who are not the host, to make it as easy as possible for players to use. Finally, I tweaked and refactored our lobby code to make sure that it is stable. In total, this all took about two weeks.

## **Mad Bomber Vehicle**

I designed and created the Mad Bomber vehicle in its entirety. I made the model for the Mad Bomber vehicle in Autodesk Maya, making sure that my design would work well in mesh deforming collisions. The mudguards are thin so that they will crumple, and the main body is rectangular and has few polygons so that regular subdivisions can be added. This took about a week.

## **Deformations Prototype**

I worked alongside Jed to create the first iteration of our soft body deformations. It was a proof-of-concept that meshes could be deformed from any hit location and that the deformations could be altered from a given set of parameters. This took about a week.

## **Debugging and Support**

As I have a large amount of previous experience with Unity, I spent a significant proportion of my time helping my teammates with bug fixing and generally getting to grips with Unity and its quirks. I also helped build the physics engine to allow testing on Windows whenever changes were made. In total, I would estimate this all to have been about two weeks' worth of work.

# George Sains

*Focus: Asset modelling, Shaders*

## **Camera Programming**

For the MVP, my primary focus was creating a controllable camera for the vehicles. This used multiple Cinemachine cameras and a master camera which would select an appropriate angle. This allowed the user to either look ahead or use the mouse to look around, as well as having an automatic reverse camera. This camera was the basis for the AI camera in the final product. This section took around two weeks.

## **Gameplay programming**

Later in the project I developed the functionality of the game-mode included in the final product, which we named Capture the Gubbinz. This works similarly to our other pick-ups already in the game; however, it does not respawn and is dropped on death. It also provides points and ultimate charge to whoever is holding it. This took two weeks to develop.

I made one contribution to the in-game UI, which was to add team names above the vehicles. This used a TextMeshPro object which will automatically rotate towards the user's camera. I also added a function which changes the colour of the text when that vehicle picks up the Gubbinz to add another way of identifying which vehicle has it. This feature took about a week to develop.

## **Shaders & postprocessing**

One of the main contributions I made was a custom shader to be used for our assets. This was a fragment toon shader to give our game a fun and unique look. Later in development I made a variant of this shader to be applied to vehicle which had the Gubbinz. This is identical, except it can be seen through walls as a block colour. This is used to help other players find whoever has the Gubbinz and track them down. All this together was around three weeks' worth of work.

A similar contribution I made was adding postprocessing to the game. Initially this was just a few effects such as vignette and colour grading, allowing us to give the visuals more depth and a brighter tone. When the cave was added, we decided to create multiple postprocessing volumes to our map which gave us the ability to create separate sections of postprocessing. I worked on this new section, giving a cooler colour palette to the inside of the cave. I spent one-week implementing this.

## **Vehicle modelling**

I designed and developed a vehicle for the game. This design was more experimental, as it was a bike with a sidecar. This came with many challenges with design and with implementation, such as only having 3 wheels. Our vehicle script worked with 4 wheels, so I implemented an invisible 4<sup>th</sup> wheel to stabilise the car, while still giving it an asymmetrical functionality. This vehicle was not included in the final product however, as playtests showed that the complexity of the model did not translate well to the mesh deformations. We decided that we wanted to focus on vehicles which showed this off best, and so removed the bike. I spent a week working on this.

## **Level Design**

I created the cave section in the final version of the map. I did this by extruding sections in one of the cliff edges in Autodesk Maya. I then edited edges and faces to give it more detail. I developed some lighting and assets for the cave such as glowing mushrooms and stalagmites. This section took about a week to make.

Throughout the development process I modelled several assets for the levels, such as ramps and rocks. I kept our game's low-poly style in mind in order for the models to fit with the setting of our game. Across the project I spent roughly one week of development time on these assets.

# Software, Tools, and Development

---

## Overview

Our game is made using the Unity game engine. This software is free for students and small developers. Within Unity, we have made use of various free plugins to either enhance the gameplay experience or improve the tools we had to work with. The most significant of these is Photon Unity Networking, which is central to our networking and matchmaking services. We also used TextMeshPro to render UI elements, Cinemachine for advanced camera features, and the “FREE Skybox Extended Shader” by BOXOPHOBIC for our skybox [2][10].

Outside of Unity, the second major creation tool used has been Autodesk Maya for professional 3D asset modelling. To create the voice chat, we made use of the PeerJS library and the Coturn TURN server project [11][12]. To manage our remote GitHub repository, we used a combination of GitKraken and GitHub Desktop [13][14].

## Unity

From the beginning of this project, we have been using the Unity game engine [15]. This was the obvious choice as more than half of the team already had experience in the engine and it offers a rich feature set. It allows developers to export to the web with its WebGL 2.0 implementation [16]. Unity’s support for plugins was also a feature that drew our attention. One invaluable tool we have made regular use of is the built-in profiler. It provides information on CPU, GPU, and memory usage. Memory conservation has been a challenge, so we use this feature regularly. WebGL 2.0 has a hard limit of 2GB of memory [17], which we have exceeded several times.

Photon Unity Networking (PUN) is a networking solution that provides matchmaking services to Unity games [18]. It allowed us to get up and running with fully functioning multiplayer within the first two weeks of the project. Its simplicity has allowed us to focus far more on our key technologies, giving us more time to polish the networking experience rather than build it from the ground up. It provides remote procedure calls (RPCs), which allow for procedures on one player’s instance of the game to be called from another player’s instance. These made synchronisation and data transfer between players much easier.

Unity’s own TextMeshPro plugin renders text for our UI and HUD elements [19]. Profiling revealed that it has several issues with the efficiency, particularly relating to memory allocation. This presented a problem. With careful reduced usage in high intensity scenes, we have managed to mitigate these effects. Cinemachine is another Unity-made plugin, which supports advanced camera manipulation. With our work on AI camera systems and our telecaster feature, it became a necessity.

While we have used many free assets during testing, for the release of the game we are only using three assets not created by the team. Two components are the skybox and engine noises, which comes free in the Unity asset store [2][3]. The final is the typefaces used in our UI [4][5]. Our reasoning is that these are small components which take a considerable amount of time to create without expensive tools.

## Autodesk Maya

All 3D models in the game have been created in Autodesk Maya [8]. Maya is an industry standard tool and has been used in at least 18 films winning the academy award for best visual effects [20]. It can export models in a variety of standard file formats, making importing models into Unity a simple affair. Throughout the development process, there has been a lot of discussion between designers and programmers about how models should be made. This is particularly important for our game as our mesh deformation scripts have very particular requirements about vertex density and model shape. Vehicle models were required to not have many vertices, and large flat surfaces allowing for effective deformation. The bodies also had to be made from one single mesh, as our soft body script will need to travel across the entire mesh using a breadth-first search. If there are any disconnected vertices, they will not be reached and therefore will not be moved in the deformations. Vehicle bodies were made by inserting edge loops onto a basic rectangular cuboid and moulding these new vertices to fit the shape of reference photos placed on the axis.

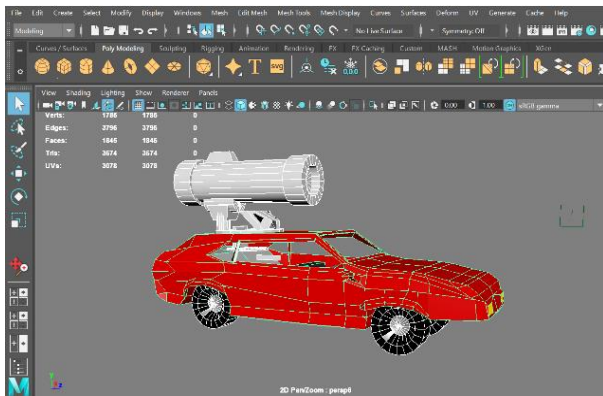


Figure 11: “The Interceptor” vehicle model in Autodesk Maya

When constructing our maps, we first used Unity’s terrain making tools to construct the basic desired shape. We were then able to convert the terrain into an object using open-source code [33], and these objects could then be manipulated in Maya. We used the reduce tool to reduce the number of vertices in our model by around 90%, helping us achieve the desired low polygon look. However, this tool often causes shapes in the model to be deformed, so we carefully organised the remaining vertices into our desired map shape.

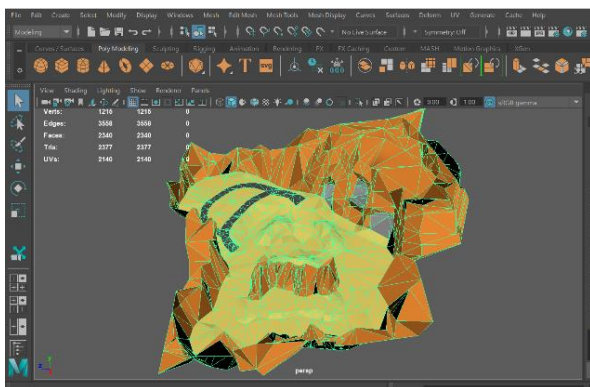


Figure 12: The “Crash Valley” map model in Autodesk Maya

## PhysX

PhysX is an open-source physics engine developed by NVIDIA [21]. It is the physics engine used by the Unity games engine for all its 3D physics simulation. We realised while developing our soft body physics that Unity did not provide enough control over the physics engine, so we switched to an external build of PhysX, rather than the one inside Unity. We opted to continue using PhysX because this physics improved the look and feel of our game. We did not want to switch engine and spend weeks tweaking our vehicles to make them drive well again.

## Software Maintainability & Git

In a project as large as this, code maintainability is a serious consideration. We have consistently built software with modularity as a guiding principle. One clear representation of this is our vehicle management system. Simply by putting a vehicle prefab in a selected folder, it will be automatically displayed in the lobby, and a dummy version will be created for menu scenes. The vehicles themselves are built with separate components. Any weapon could be attached to any vehicle with no manual changes for example. There are many other similar instances including the lobby being built to accept multiple different options for the map and the vehicle driving interface being designed to accept varying forms of input.

One significant time investment for us was in our gamestate tracker, the final version of which is explained in the technical content section (page 20). Until the MVP, we had been using our first iteration which suffered from many problems including performance and code safety issues (primarily due to networked race conditions between different players). Rather than patch it as we go, we decide to rewrite this component from the ground up, using true best practices and making intricate performance improvements. This process took approximately a week, during which time little progress could have been made on the rest of the game, as a lot would have to be adjusted once the new changes came into effect. We fully believe that this up-front time allocation was worth it, as we had very few issues moving forward. The decision was made with long term stability in mind. It was therefore made to be expanded upon in the future. Later developments, such as ultimate abilities and the Gubbinz, have all been added to the gamestate tracker with no development delay.

The entire project has been hosted in a remote GitHub repository [7]. Each member of the team has used either GitKraken or GitHub desktop as a git GUI [13][14]. When we want to add a feature, the first thing we consider is what branch to use. If we are building off previous work, we may bring an old branch up to date, then work within that. Otherwise, we make a new one and use it exclusively for development on a single subject. Once the changes have been made, we first merge main into the branch, resolve any conflicts, test, and finally merge back into main. This strict path minimises issues arising from merge conflicts and theoretically maintains a working build on main every time.

## Testing

Testing is a key part of any software development team process. For the most part, we have kept testing in-house. We had regular weekly group playtesting sessions, where we would play the game together and look for bugs. This is mainly due to having regularly brought in sweeping changes to large technologies, so we had been looking for bugs rather than player feedback.

To make the testing process easier, we made unit tests for the gamestate tracker, which all the rest of the game logic is built off. Having these unit tests meant that we could guarantee the tracker was almost entirely bug free, significantly reducing the scope for errors in our game logic. These unit tests covered creating messages, serializing them for transmission and ensuring that they were correctly applied when they reached their destination. The tests also cover handling conflicting edits and ensuring that observers are notified after edits are made.

To ensure any edge cases are covered, we made a system to make randomised edits for our test cases. The tests could then be run hundreds of times, so we could be reasonably sure that the system would work regardless of what edits were made. We seeded the random number generator used so that it would always make the same edits, making fixing any errors detected by the tests much easier.

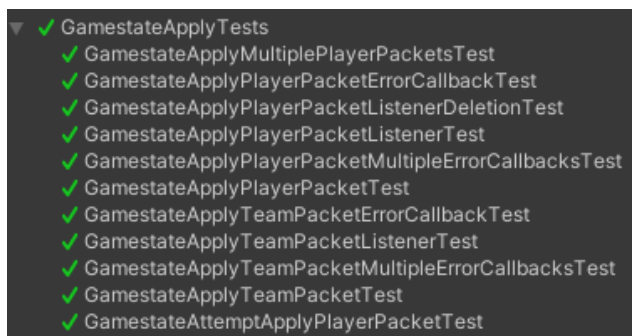


Figure 13: Some of the unit tests for the gamestate tracker

As we are only a 6-person development team, we can only test so much on our own since our game is designed to function with up to 12 players in a single match. In the closing stages of the project, we have gathered feedback from over 20 users in a range of group sizes to polish the gameplay experience. Evaluations were collected with a Microsoft form, and exported to Excel. This gave us a very clear idea of what users liked, disliked, and were confused

by in the game. The decision to change our game mode from a checkpoint race to a points-based system was led by comments from players during early stages of testing.

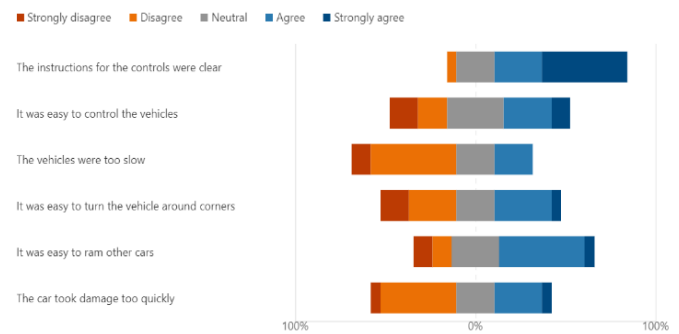


Figure 14: Feedback via Microsoft forms on user experience as the driver

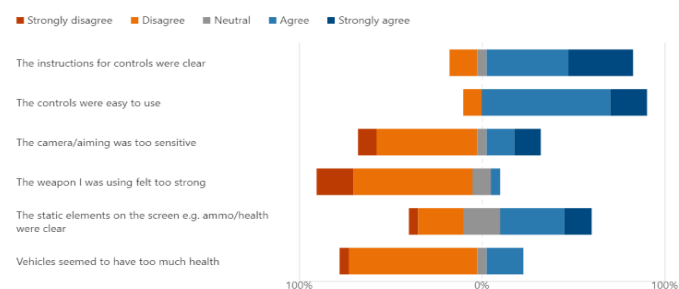


Figure 15: Feedback via Microsoft forms on user experience as the gunner

As computer science students, we had easy access to a wide array of like-minded individuals. This meant that a lot of our subjects had an above average amount of experience playing video games. This was extremely helpful in some respects, as some testers were able to give extremely detailed comments and ideas. The downside is that we only had feedback from a very focussed demographic, and we could potentially suffer in fields such as generality and accessibility.

We expanded our search to include a wider array of personalities and backgrounds from ages 17 to 54, to seek new perspectives on our work. We paid particularly close attention to feedback from individuals outside of our own age range and experience level, to maximise approachability. 22 responses from these tests, some of which are shown in Figures 14 and 15, lead to improvements in UI, better in-game tutorials, and therefore clearer objectives. One of the most significant improvements was our auto-ram feature which brings a key component of our game down from being one of the most challenging things to learn, to being a simple button press.



### Network

Insane Ian uses the Photon Unity Networking (PUN) library to synchronise the game between players [22]. PUN uses a client-server architecture, with a relay server to exchange data streams and remote procedure calls (RPCs) between players. RPCs are a system that allows players to call certain procedures on each other's game instances. Our structure employs an authoritative master framework, with client-side physics simulation. This model involves holding the environment currently on the master client instance's game as the 'true' state. Critical game updates are passed to the master client, processed, then the result distributed to other clients. This model of network architecture is used over alternatives such as full authoritative server, due to the expense of running a full dedicated game sever opposed to a relay server, and the lockstep model (multiplayer framework in which player inputs affect multiple independent, concurrent game simulations) due to latency issues and input lag. Matches of Insane Ian are contained within Photon's *rooms*. Each room is housed within a lobby, and each lobby is hosted on a remote relay server.

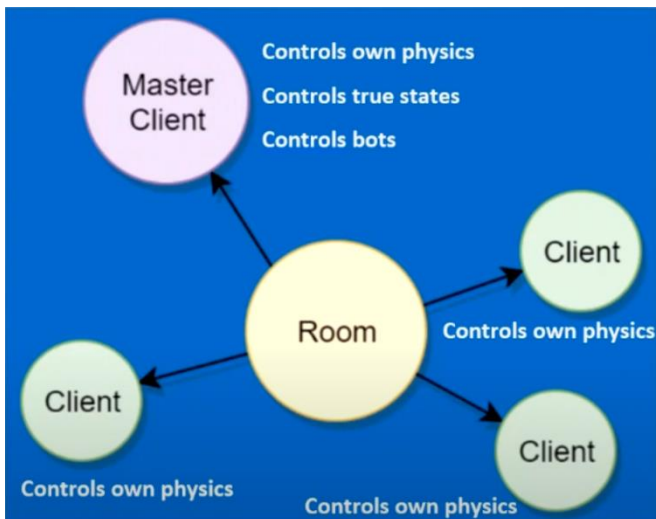


Figure 16: A network diagram demonstrating the master client framework

### Vehicle Synchronisation

Our motivation for the network design was to have rapid synchronisation and tolerance for latency, as well as allow for detailed networked physics interactions. Our solution involves synchronising the velocity and angular velocity of players, rather than their position and rotational data. This allows networked vehicles to be simulated fully

within the physics engine. This means vehicle movement is smooth and tolerant to the network send/receive rate. The fact that vehicles have their physics simulated fully locally allows for collisions to be modelled smoothly, without lag, and in full detail. Treating vehicles in a naïve manner (merely updating positions and rotations) results in a choppy and unrealistic movement. This is even the case with interpolation applied, since the network cannot guarantee an exact packet receive rate. Furthermore, in the authoritative-master framework, if collisions are not modelled locally then vehicles can appear to phase into each other, due to how the resolvent forces from physics calculations must be sent over the network and back.

Each client keeps a record of the 'true' state values for their vehicle. Due to network latency and differences in computation speed, desynchronizations can occur. To account for this, the true value of a vehicle's position is also transmitted. If the vehicle's position and rotation is too far from the true value, the vehicle is moved back to the correct position. The data streams are sent via UDP, as it has a low latency, and any errors are likely to be lost in the physics simulation.

The tolerance in this transmission method allows for vehicles to appear to move smoothly, even in a lag spike, as their motion will continue to be carried out through physics. In collisions, we took advantage of this fact by slowing down the local timescale during a crash, allowing involved players to see the deformation in higher detail.

### Gamestate Tracker

The game needs to store and synchronise data about both individual players and the teams they belong to. This is handled by the gamestate tracker, a network synchronised database. The motivation behind creating the gamestate tracker was to ensure reliability. At the start of a game, setup is a surprisingly complex procedure, since the game must retain and synchronise data about who is in which team, the role of each player, as well as which vehicle is owned by which team.

Before the creation of the updated gamestate tracker, there were many race conditions with transmitting this data. This would lead to game breaking bugs, such as players being assigned to the wrong teams, assigned to the wrong vehicles, or being left out of the game entirely.



The gamestate tracker supports much of the functionality of a normal database. It allows player and team entries to be created, read, updated, and deleted, and synchronises their states between players. It uses bitpacked messages for synchronisation. This means the messages are designed to take the smallest number of bits possible, minimising their bandwidth use. This ensures the tracker has minimal overhead while operating in the game.

To prevent conflicting edits being made to the entries, each entry has a revision number that is incremented each time it is edited. If two edits are made with the same revision number, it is detected as a conflict and only the first one is used. An error handling procedure is called to allow the game instance that made the second edit to handle the failed edit gracefully.

The gamestate tracker makes use of the observer pattern to allow other parts of the game to be notified when entries are edited. When another part of the game registers itself as an observer of an entry, it passes the gamestate tracker a function that is called whenever the entry is updated. This minimises unnecessary reads to the tracker, as reads are only made when the required data has changed.

We designed the gamestate tracker to be thread safe; each entry has a mutex to prevent multiple threads from accessing it simultaneously. We realised this was unnecessary for WebGL builds, as multithreading is unsupported [23]. However, should we wish to develop the game into a standalone application in future, it would be easy to synchronise state across multiple threads.

## The Driver and Gunner

The driving controls are entirely governed by the physics engine; they accelerate by applying torques to the PhysX wheels and steer by turning them. The wheels have parameters governing suspension and tire friction behaviour. The suspension is modelled as a single spring for each wheel, where damping rate and strength can be manually set. We have spent significant time tuning these parameters to provide a fun and unique driving experience for each vehicle in the game.

We found during our testing that these parameters offered a very realistic control model, however this was not ideal for our game. Vehicles tended to roll in tight turns, particularly over rough terrain. To combat this, we took inspiration from the real world and modelled an anti-roll system. This measures spring compression on one

side of the vehicle and exerts a proportional downward force on the other side, to maintain stability.

Gunners can rotate their turret by setting a target position vector. The turret itself follows this target, if outside of a deadzone (small rotations for which the turret body need not update).

When a weapon is fired, the target point is calculated. This target point is determined by firing a ray from the player's camera. If the turret is outside of the deadzone, then the ray is instead fired from where the turret is currently pointing. This causes the shots to be very accurate when the turret is stationary, but inaccurate if the turret is rotating rapidly. We sought to include this feature for two main reasons. Firstly, we wanted to increase the level of depth to the gameplay of the gunner, requiring them to take a moment to aim for an accurate shot. Secondly, the gunner's rotation is streamed and interpolated across the network. Implementing a traverse speed effectively reduced the jumps in turret rotation that could normally be exhibited with fast mouse movements.

Controls for both driver and gunner use an interface so that both human players and AI agents can control vehicles in the same way.

## Weapons and Damage

Weapons are designed as a polymorphic class, with public methods designed to be called by a weapon manager. Weapons come in three predominant types in Insane Ian: Projectiles, Hitscan, and Beams.

All weapons rely on the same principle of interaction with other vehicles. When a weapon hits a vehicle, the owner of the weapon calls an RPC on each player's instance of that vehicle, causing it to take damage.

When a weapon is fired, a deviation is applied to the shot, altering its direction by a small amount. This is accomplished by re-calculating the target vector to a random position inside a scaled sphere about the initial point. When a target is hit, the damage is modified as a function of the distance the projectile travelled; this is performed as a matter of game balancing, since we wanted weapons to be more effective at close range to encourage vehicles to stay together to increase the likelihood of collisions – thus showing off our soft-body deformations.

### Projectile Weapons

The first designed weapons, projectile weapons, instantiate a physical object upon firing. Collision of this object with any vehicle will immediately apply damage via RPC. The gunner fires a 'true' projectile, dealing damage if their instance of the projectile hits. To account for latency in the firing of the projectile, other players simulate the distance the projectile would have already travelled based on its firing time. They then create the projectile as if it had already travelled this distance.

### Hitscan Weapons

Hitscan weapons do not create physical projectiles. Instead, they fire rays, which hit the target instantly. This makes any difference in firing time much more noticeable, so lag compensation is trickier.

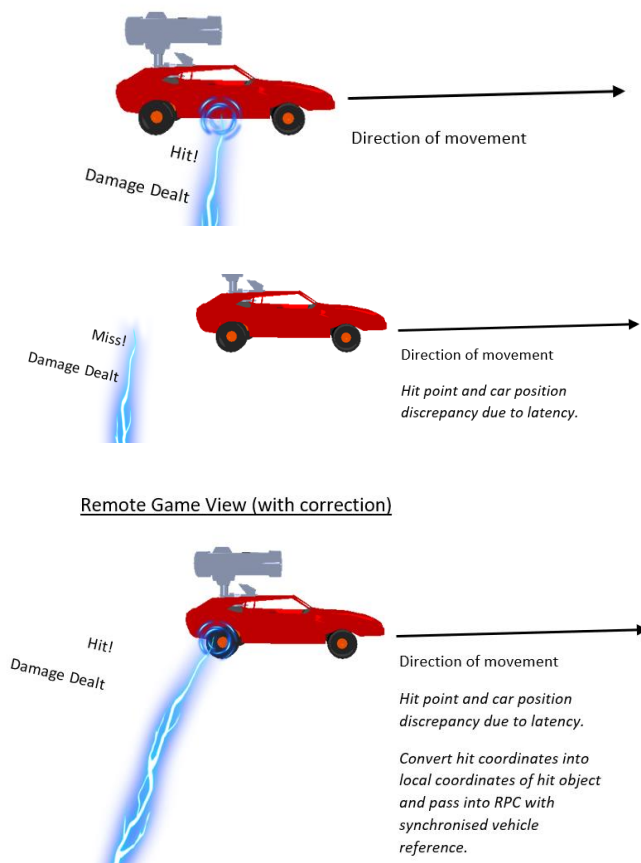


Figure 17: Diagram illustrating tracer hit correction

An issue that had to be faced was the discrepancy in the position of moving targets and the firing visualisation. For the other players, the shots would not appear in the correct position, so would appear to miss while still dealing damage. To fix this, we developed a technique coined tracer hit correction. In tracer hit correction, if the ray intersects with a vehicle, it returns the local coordinates of the hit point in relation to the hit object, as well as a reference to the hit vehicle. The weapon

effects are handled via RPC on other game instances to convert the reference into a transform and local coordinates into world coordinates, then play the relevant particle effects. The result is, if it looks like your vehicle got hit, then you did. If you shot an enemy vehicle, then you also did – regardless of latency.

A second issue we encountered was the substantial performance decrease experienced when a gunner was using a rapid-fire weapon. This would cause a barrage of RPCs to be sent over the network. An optimisation was developed to mitigate this issue. If a weapon is deemed to be rapid fire, its graphical effects will be simulated locally using a remote-firing coroutine. For other players, the weapon will continually fire non-damaging shots in the direction it is facing, with the firing player periodically sending the damage dealt. This significantly reduced the number of RPC calls needed, saving valuable network bandwidth.

### Beam Weapons

Beam weapons were created to demonstrate the quality of particle effects we have implemented in Insane Ian. In terms of dealing damage, beam weapons function very similarly to hitscan weapons. However, it is the graphical effects that warrant a technical explanation. Beam weapons are notoriously difficult to implement in online multiplayer games.

A beam is composed of a start particle effect, an end particle effect, and a line renderer with a scrolling material. When firing, the beam sets the start and end of the line renderer to be at that of the gun and the hit location, moving the particle effects appropriately. On the gunner's instance of the game, this operation is called every frame update. To achieve an adequate effect over a network however, multiple remote coroutines are used.

Two objects are created to store the start and end points of the beam. The gunner regularly updates the location of these over the network. When the beam is first fired remotely, only its first hit point is defined. This calls a coroutine to linearly interpolate the end point of the beam from its origin to hit point as time passes. During continuous firing, the beam hit points are updated regularly, calling coroutines to interpolate the beam's hit position. If the beam hits a moving target, the aforementioned *tracer hit correction* algorithm is applied.

## AI

As our game is played in teams of 2, it was essential for us to implement both driver and gunner AIs to cater for single player teams, as well as to fill out player slots to stop the game from feeling empty. Due to the efficiency of our game, each match can contain 12 people and still run smoothly. Having the AI means that users can have a fun experience, regardless of the number of players.

### **Driver AI**

One of the challenges to overcome was to create competent AI drivers that did not rely on Unity's inbuilt navigation mesh, as this does not support physics-based movement. The driver AI was written in a way that capitalised on the player driver control interface. The driver AI is, in essence, a script that emulates the actions of a player by calling control methods ordinarily actioned using key presses – such as acceleration and turning. They are run on the master client instance and from the perspective of other clients, act the same as players. The bots follow an interpolated path between a set of AI marker waypoints which mark points of interest. Bots will attempt to follow this path using their on-board decision-making systems, steering towards it as they veer off-course.

Obstacle avoidance is implemented via a suite of sensors which return object telemetry ahead of the car. A set of PhysX raycasts are fired at each physics update, returning a set of calculated values such as estimated time to crash and amount to move left and right. These values are fed into the AI algorithm, to control its decision making. The length of these sensors scales linearly with its forwards velocity, such the car is always able to detect targets up to a second ahead of time to take evasive action. For instance, if the sensors detect a static obstacle ahead and right of the car, the AI will decide to steer left. If steering left is not enough (determined by a reducing estimated time to crash), then the AI will decide to brake. Objects can be marked using the PhysX layer property on their collider such that the AI responds differently to them. Players are marked differently to rocks, and an AI will actively seek to collide with a player if they are detected ahead.

### **Gunner AI**

In comparison to the driver AI, the gunner AI is relatively simple. The gunner AI searches for nearby targets in view, then interfaces with the gunner weapon manager to fire the equipped weapon. To prevent the AI from being too

powerful due to its spot-on accuracy, it has a base targeting deviation applied to all its shots. In addition, there is a further deviation applied to a target based on its projected velocity perpendicular to the camera. This simulates the difficulty a player has in shooting an opponent who is moving across their screen.

## Smart Camera

Physics deformations are shown off using our smart camera system. Sensor telemetry similar to that of the bot drivers is fed into our specialised camera control system. This system is a reverse engineered variant of Mechanim, Unity's state machine animator. By overriding the base animation class with our own set of behaviours, a set of camera states and their transitions can be easily defined and visualised in the same interface. This allows the driver camera to have a rudimentary form of memory, as well as to allow for its behaviours to be adjusted in response to testing. Camera transitions can be triggered by reaching an estimated time to crash threshold, achieving a certain speed, colliding with the environment, or colliding with other players from various angles. The result is, when you have a collision, the camera reacts to show the deformation in the greatest possible detail.

The camera itself was created using the Cinemachine framework. This technology allows for multiple virtual cameras to be created, without the overhead of having to render their view. The real camera can then be switched to show the view from any of these virtual cameras at any point [10].

## Voice Chat

When we first set out to integrate voice chat, we attempted to use the obvious choice for our project: Photon Unity Networking has an add-on called Photon Chat [18]. However, we soon came to realise that this makes use of multithreading to decode incoming data streams, which Unity's WebGL build pipeline does not currently support [23]. Further research into Unity-based voice chats also hit dead ends as the solutions either did not support WebGL or had not been maintained for years and therefore would not work on recent versions of the game engine. Eventually we decided that the best viable approach would be to create a separate voice chat system using JavaScript and then control its functionality from inside the game.

### Peer-to-peer communication using PeerJS

To establish browser-agnostic communication, we use a technology called Web Real-Time Communication (WebRTC). WebRTC allows web browsers to stream audio and video media, which is exactly what we need for a voice chat. We thought that there was no point in reinventing the wheel, and so decided to use a JavaScript library called PeerJS [11]. We chose this over other libraries as it is lightweight and does not force any extra features which we would not need, therefore reducing the complexity of our program. PeerJS utilises a browser's WebRTC implementation to set up peer-to-peer connections between clients. This connection can then be used to send and receive streams of data, in our case audio media streams to allow players to talk to each other. As a person's IP address could be considered private information, we opted to host our own instance of a PeerJS server so that we can ensure that a player's data is never distributed elsewhere. This server is hosted on a Google Cloud Service as an App Engine [9].

### TURN/STUN servers with Coturn

If two players can directly access each other's internal addresses on a network, then our voice chat works flawlessly with just the PeerJS server. However, this certainly will not be the case for our panellists on demonstration day. Everyone will be connecting remotely, and therefore will be behind a symmetric Network Address Translation (e.g., a router). To solve this problem, we needed to use the Interactive Connectivity Establishment (ICE) protocol, which generates media traversal candidates that can be used by WebRTC. PeerJS already has support for this scenario, but it requires us to set up our own TURN (Traversal Using Relays around NAT) and STUN (Session Traversal Utilities for NAT) servers to route our user's peer-to-peer connections successfully. A STUN server allows any device to determine an IP address and port that they have been allocated by a NAT. This allows our players to establish connections between each other's browsers. The TURN protocol is an extension to STUN that allows clients to send data through an intermediary server. So, by using TURN and STUN in combination, our players can successfully connect to each other, even with routers and firewalls in the way.

Free TURN servers do not exist, so we set up our own. We used the Coturn TURN server project as it is well-maintained, free, and open-source software that is relatively simple to use [12]. Our instance of Coturn is

hosted on a lightweight Linux Virtual Machine provided by Google Cloud Services [9].

### Communication between C# and JavaScript

To properly integrate voice chat into our project, we needed to communicate between the JavaScript that controls our PeerJS connections and the C# that our Unity project runs on. Unity documentation recommends doing this by writing a JavaScript library that can be loaded as a plugin upon launch [24]. However, we quickly figured out that this considerably slowed our workflow. As our voice chat only works in a web build, it could not be tested in editor. This would mean that each time we wanted to test a change, we would need to build and then upload our entire project. If any changes are made to our plugin, all the WebGL build files would need to be rebuilt, which could take upwards of fifteen minutes. Instead, we found that if we wrote our JavaScript as inline script inside of a custom WebGL HTML template, only our script would need to be changed for a new build, rather than recompiling the entire game. This allowed us to develop and test changes to the voice chat much more rapidly. Of course, we still needed to communicate between this inline JavaScript and Unity's C#, so we have a much smaller intermediary JavaScript library that can pass data between the two. The functions can then be accessed using `[DllImport("__Internal")]`. Although this adds seemingly unnecessary complexity to the communication process, the performance impact from this more roundabout approach is minimal to the game; and as it allowed development to be much faster, we feel that this decision was worthwhile.

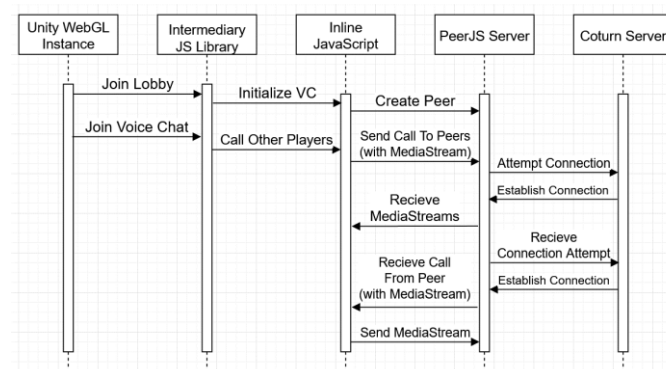


Figure 18: UML sequence diagram of voice chat behaviour

### Gameplay Integration

We have implemented voice chat primarily to enhance the teamwork involved in our multiplayer gameplay. While in the lobby, a player can talk to everyone else that has connected to the chat. To encourage communication

within teams, we then mute all but a player's partner while the game is in progress. To achieve this, we store a reference to each player's audio source alongside their unique voice chat ID so that it can be muted/unmuted when necessary.

Peers in PeerJS need to have unique IDs to work correctly. To ensure this, we decided take advantage of the unique data that already exists in our game. Photon Unity Networking lobbies require that each room has a name as a unique identifier, and within these rooms each player is given a unique actor number. By concatenating this room name and actor number together, we form a string that is guaranteed to be unique across all instances of our game. This is what we use as our voice chat IDs.

## Direct PhysX Access

The Unity games engine uses NVIDIA's PhysX physics engine internally for all its 3D physics simulation [25]. It provides a wrapper around it, with support for most common physics simulation features needed in games. Unfortunately, one of the things it does not have inbuilt support for is soft-body physics, where colliding objects bend and crumple. It only supports rigid-body physics, where objects remain the same shape and simply bounce off each other.

The most common way to simulate soft-body physics, which would be possible using the physics components Unity provides in its wrapper, is to form the overall object out of many small point objects, connected by spring joints. This produces realistic soft body physics at the expense of performance. Games that make use of this technique typically require reasonably powerful computers to keep up with the physics simulation. These games will also have optimisations in place to make the joint simulation as fast as possible, which Unity is unlikely to have as its joints were not designed to be used in this way. Also, we had no guarantee that people playing our game would have access to the sort of processing power required for this.

Instead, we opted to use the standard rigid-body physics and reduce the impulses applied during collisions. This makes the objects pass through each other slightly, which, combined with graphical deformations, produces the effect of objects squashing into each other. This technique is much more performant, requiring little more processing power than the standard rigid-body physics simulation.

### Limitations of Unity's PhysX Wrapper

The obvious way to reduce the impulses applied during a collision is to apply inverse impulses. This does not work for two reasons.

The first is a limitation of Unity's wrapper. It does not provide all the information about the impulses applied in the collision; it only provides the sum total impulse [26][27]. Applying the inverse of this total impulse would reduce the linear impulse applied, but not any torque. This would produce significant spurious rotations in the vehicle during collisions.

In Figure 19 you can see an example of this. As Unity does provide the locations where the impulses were applied, it would be possible to counteract the impulse correctly in this example by applying an inverse impulse from the location of the collision. As soon as multiple impulses are involved, which is normal for the physics engine, this would become impossible. This is because there is no way of knowing what size impulse should be applied at each location given just the sum impulse, so it is hard to do better than just applying the inverse sum impulse from the centre of mass.

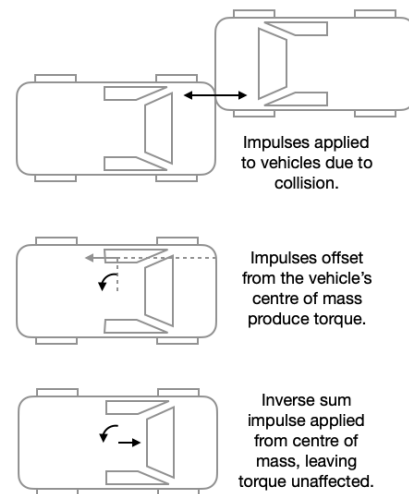


Figure 19: Torque produced during collisions because of naïve impulse counteraction.

The second is a limitation in the way PhysX performs its physics simulation. During each simulation step, first the impulses produced in any collisions are computed, then the positions, rotations and velocities of objects are updated [28]. There is no way of applying forces between these steps. This means that, even if the inverse impulses are applied, the collision will immediately be resolved



again and will apply forces to produce a reasonable rigid-body collision. The overall effect of this is to cause the vehicle to stick to whatever it has collided with and shake rapidly.

Instead, we opted to prevent the collision forces from being applied at all, while making sure that the forces were still calculated. This meant that we would be able to apply the reduced collision forces directly. PhysX does not support preventing collisions from being applied whilst still calculating the forces [29], so we decided to use a system where two scenes are run in parallel, one with collisions and one without. This would allow us to take the collision forces from one scene and apply them to the other. See the Producing Soft Collisions section for a more detailed explanation. Unfortunately, this still left us with the problem of Unity not providing the individual forces applied in the collision, so we would still get significant rotations. We also would need to work with making Unity simulate two scenes in parallel. It does have support for this but does not give us control over the order of simulation [30], so we have no idea which of the two scenes will be simulated first. This could potentially make the physics very unstable.

We decided that the best solution would be to access the physics engine directly, so that we could extract the collision forces ourselves and simulate two physics scenes without creating a second scene in Unity.

### ***Accessing PhysX***

PhysX is written in C++, while Unity only lets you use C# for programming your game. C# does have support for loading external libraries using the `DllImport` attribute, so we were able to compile and load a small C++ library that just provided wrappers for the PhysX functions for the Unity editor.

For web builds, the process was somewhat more complicated. When Unity makes a WebGL build, it first compiles the C# to Microsoft's Common Intermediate Language. This happens when C# is compiled normally, but it would normally then be compiled to machine code, so that it can be run. For web builds, Unity instead converts it to C++, then compiles that to WebAssembly using the Emscripten compiler toolchain [31]. WebAssembly does not support loading external libraries, so Unity instead allows you to give it C++ files to add in just before the compilation to WebAssembly [30]. This means the C++ functions can be accessed using

`[DllImport("__Internal")]`, which tells the program to use itself as a library.

Trying to make Unity compile the PhysX source code in this way causes the compiler to throw a large number of symbol multiply defined exceptions. We realised this must be because it was including its own instance of PhysX in the same way, so was defining every PhysX class and function twice in the same C++ file. This meant we could access Unity's instance of PhysX from our C++ wrapper code simply by providing it with the right header files. This allowed us to reduce both the time spent compiling, as the PhysX code base is very large, and the size of the final build.

### ***Replacing Unity's Components***

The components Unity provides to control physics simulation in the game are for the most part very similar to those provided by PhysX. This meant it was reasonably easy to make our own physics components be simple wrappers around PhysX's components, and be drop-in replacements for Unity's components. Making them in this way allowed us to continue working on other parts of the game using Unity's physics components, confident that the transition would require little or no modification of our code.

### ***Producing Soft Collisions***

To produce soft collisions, we created two physics scenes in parallel. The first one, the main scene, has all the physics objects in it and standard rigid-body collisions. The second one, the ghost scene, only contains the objects that we wanted soft-body collisions for and has no collisions.



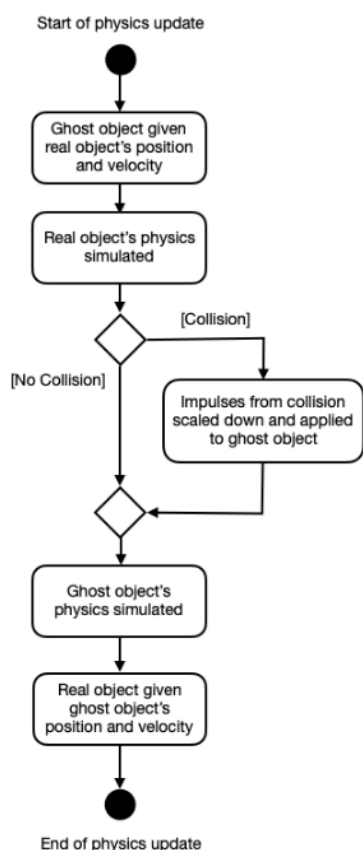


Figure 20: Flow diagram of a physics update

During each physics update, each ghost object is first given the corresponding real object's position and velocity, to synchronise the two physics simulations. Then the physics in the main scene is simulated. This means that any collisions in the main scene are resolved and all the objects in it have their positions and velocities updated.

As part of the simulation, PhysX returns all the impulses applied to the objects during any collisions that occurred. These are then scaled down and applied to the corresponding ghost objects. This means the ghost object receives some, but not all, of the force of the collision, causing the collision to slow it but not stop it.

After all impulses have been applied the ghost scene's physics is simulated, then the real objects are given the positions and velocities of the ghost objects. This means the real object is also slowed but not stopped by the collision, passing through whatever it has collided with slightly. This causes the collision to be much softer. See Figure 20 for a diagram of the process.

### Scaling the Impulses

Scaling the impulses by the right amount is crucial to deliver satisfying soft-body collisions.

The impulse scale governs the effective compression resistance for our simulation. An impulse scale of 0 is no compression resistance; the objects will pass through each other without slowing down. An impulse scale of 1 is complete compression resistance; the objects will not pass through each other at all.

In a real collision, the resistance to compression becomes greater the more compressed the objects are. The greater the resistance to compression, the closer the collision gets to a rigid-body collision. That is, a collision where the objects have complete compression resistance.

To simulate this, we changed the impulse scale based on the depth of penetration. That is, how much the two objects intersect in the collision. A deeper penetration means the objects would have compressed more in a real situation, so they would have a higher resistance to further compression. Hence a deeper penetration means a larger impulse scale.

Although this produced realistic soft collisions, we wanted them to be slower and more visually impressive. We decided the best way to do this was to increase the time spent at low speeds during the collision. We did this by reducing the impulse scale when the vehicle was travelling slowly, so that there was less resistance, and the collision would carry on for longer. This gives the player longer to process and take in the collision.

### Limitations of Soft Collisions

This technique is not perfect, and we have observed a few issues with it that are not easily fixable.

One of the most significant issues is that it is possible for the soft collisions to cause the vehicle to pass through sufficiently thin objects. This is most noticeable with the terrain, which is a thin sheet made from many triangles. As the triangles are perfectly thin, it is relatively easy for the vehicle to pass through the bottom of the map and plummet into the endless void. We prevent this for the most part by disabling the soft-body collisions when the vehicle body is in contact with the ground. It is still possible to pass through the map walls by hitting them at the right angle, however. As an additional safeguard against this, we also have a second floor that teleports vehicles that pass through the first floor back into the map.

It would be better to fix this by adding thickness to the terrain, but this would require a fundamental restructuring of the physics engine's idea of floor.

Another issue we have is that, coupled with the mesh deformations, it is possible to collide with parts of the vehicle where there does not appear to be any body. This is because the shape of the body in PhysX must be a convex polygon and cannot be changed easily. This means that any changes in the visual body shape are not reflected in the physical shape of the vehicle.

It would be possible to prevent this by creating a new convex body from the visual body mesh. However, the process of creating a new convex physics object is intensive and would slow our game significantly.

## Visual Deformations

The visual body of the car is a mesh, a polyhedron with many vertices and faces [32]. It is necessary to alter the shape of the mesh during collisions to produce the effect of the vehicle body bending from the force. Due to the large number of vertices in the mesh, it is necessary that the algorithms used to compute these deformations are fast. Any inefficiencies could potentially slow the game significantly.

The data structures Unity uses to store these meshes are optimised for size, as it only needs the meshes for rendering. It does not care about which vertices have edges connecting them; it only needs to know where the triangles are. This is information that we need to be able to calculate deformations from collisions; forces applied to a point on the mesh will be spread by the connecting edges. To make this information easily accessible, we converted the mesh from a set of triangles and vertices to a graph of vertices and their connecting edges. This graph allows a vertex's neighbours to be found very rapidly and gave us access to the various algorithms commonly used for graphs. These meshes have their vertices further subdivided in order to increase the resolution of deformations.

Before we could make use of this graph's functionality, we needed to know what deformation the initial collision caused. To compute this, we first calculate the plane that separates the vehicle from what it has collided with. This is done using information about the contact points in the collision provided by the physics engine.

The vertices are then separated into those on the near and far side of this plane. Those on the far side are

considered candidates for deformation. This first pass is used because it can be done very rapidly. Checking what side of the plane a vertex lies on can be done using a dot product. See Figure 21 for the details of the calculation.

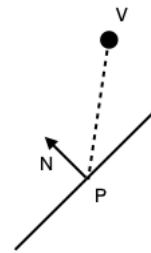


Figure 21: given a plane passing through point P with normal N, a vertex V lies on the near side of the plane if  $(V - P) \cdot N > 0$

A second pass is then performed to make sure the vertices are inside the object the vehicle collided with. This is a much more intensive calculation but can be handled by the physics engine.

All vertices that are still candidates for deformation after these two passes are then deformed. They are projected onto the collision plane, making the mesh buckle inwards at the point of collision. See Figure 22 for an overview of the deformation algorithm.

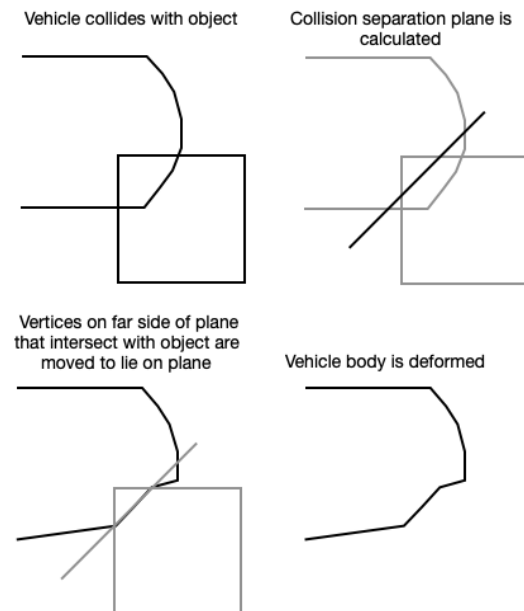


Figure 22: A demonstration of deformation

If only the vertices directly involved in the collision are moved, it is possible for some collisions to look rather unrealistic. Instead of the mesh bending around the collision, it would leave a neat hole in the shape of the object collided with.

To prevent this, after the initial deformation is made, the graph is used to dissipate the force of collision out into the rest of the mesh. A breadth first traversal is run, starting from those vertices directly involved in the collision. For each vertex, the adjacent edges are checked. If the edge connects the vertex to one that was directly involved in the collision, or one that has previously been visited by the traversal, its length is checked. If the length has increased by too much from its starting value, the vertex is moved to prevent the edge from becoming too long. Preventing this edge stretching causes the vertices surrounding the point of collision to be pulled inwards, creating the effect of the force of the collision bending the vehicle's body.

## Shaders

To give our game a unique look and feel, we decided to write several custom shaders. This meant we could fine tune a specific artistic style which could be reused across the whole game with different parameters. This would cut down on development time for making complex textures for every object, while allowing us to use simple textures to add depth to objects where needed.

### Toon Shader

The primary shader used in our game is a fragment shader. A fragment shader allows us to allocate individual colours to each pixel rendered. The render pipeline for the shader first passes through a basic vertex shader which goes through each vertex. It takes various attributes such as world position, angle, and UV data, and then passes that information to the fragment shader. The toon shader compares its own normal with the light's direction using the dot product and that will decide how lit the object is. We used a variable to decide how many sections the shader will light. As the dot product will always produce a value between  $-1$  and  $1$ , we can split up the sections across that space and group the fragments into these sections. This means all faces with a similar angle towards the light will be given a flat colour, ranged between an ambient darker colour and a lighter object colour based on the section it is in.

Next, the shader will take in the point between the view direction and the light source. It will take the dot product of this with the normal to find a location for specular light. This is then combined with the *smoothstep* function to give a single colour with a soft edge. Finally, a rim is created around the lit section of the object by taking the inverse of the dot product of the view direction with the

object's normal and then again using the *smoothstep* function for the same reason as before. This can then be combined with any texture needed, such as that of the ground of the map. For this, simple pixelated noise samples were generated for different parts of the ground and then scaled to fit the area used.

### Gubbinz Shader

We created one other shader to be used in our game. The purpose of this shader was to be applied to the player with the Gubbinz and would allow other players to see their vehicle silhouette through walls, and thus make them easier to locate. This works by having a multiple pass shader, meaning it will render it multiple times with different settings. The first pass uses a combination of settings to make it render last in the render pipeline, placing the colours in front of other objects. A fragment shader then returns a flat colour, rendering the entire object as a single tone. The second pass uses the previous shader so it will render normally if the car is not behind another object.



Figure 23: The Gubbinz shader appearing through a wall

## Optimisations

To make our game accessible to people with a variety of devices, it has been imperative that we optimise our game to make the most of the hardware that the players have available. WebGL also has a 2GB hard limit on memory use [17]. This means we must make sure that we do not allocate too much memory in our game code.

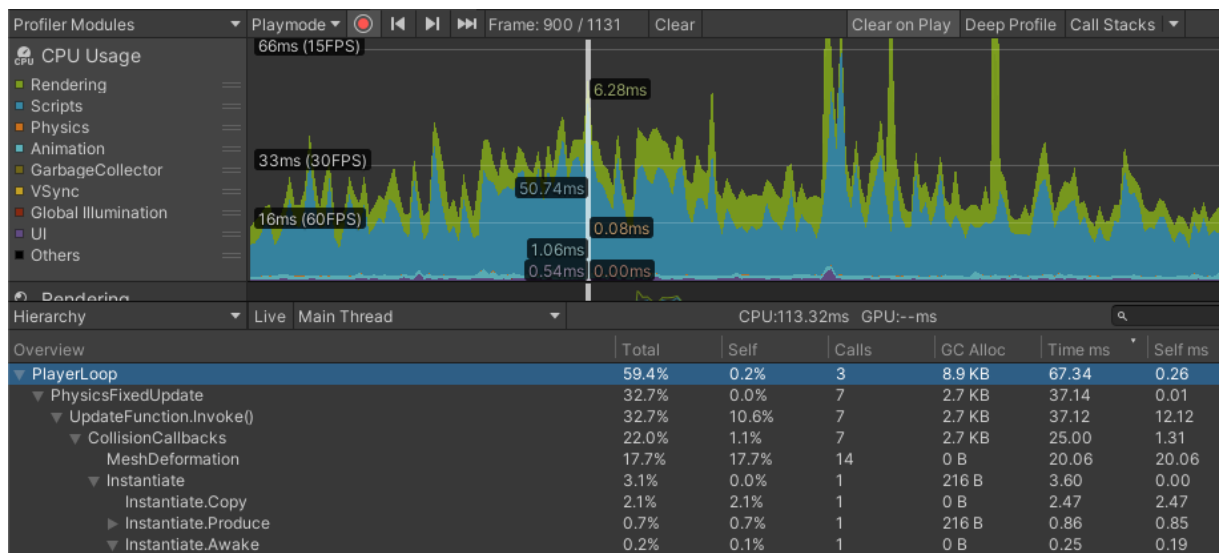


Figure 24: Results from running Unity's inbuilt profiler on Insane lan

To do this, we regularly ran Unity's in-built profiler on our game. This gives us frame-by-frame data on how our game is performing. It gives an overview of where the most time was spent in our code, as well as where memory was allocated. See Figure 24 for a sample of the data produced by the profiler.

We used the data from the profiler to identify the sections of our game that would make the best candidates for optimisation. These were usually the intensive algorithms used to compute deformations; these had many optimisation passes before they reached a level of efficiency suitable for the release of our game. Other optimisations include removing some of the more intensive graphical effects and introducing object pooling.

Object pooling is a technique to minimise runtime allocations where, instead of allocating and freeing objects repeatedly, a permanent set of objects is created. When a new object is needed, one is taken from the set. When the object is finished with, it is returned to the set. This means the garbage collector does not need to deal with any memory fragmentations caused by repeated allocation and freeing, which would otherwise regularly cause large drops in performance. Ideally, we would remove all memory allocations beyond the initial setup of the game, but this was impractical in the timeframe we had available.

The graphs used to calculate deformations for the vehicles used significant amounts of memory. We reduced this by creating a graph for each vehicle type, rather than creating a separate graph for the body of each vehicle. Vehicles of the same type share the graph but use

their own body mesh for the positions of their vertices. This means we only ever have three graphs, rather than potentially having up to six.

## Cut Content

### Telecaster

When looking for ways to emphasise collisions during gameplay and show off soft body deformations, we decided to create a telecaster. We implemented it as a small screen in the corner of everyone's view, and it would display areas of high intensity action, keeping players engaged in combat. The objective was to show off key moments happening anywhere in the game to every player, regardless of where they were. It provided both an excellent view of the action and would support players in finding their way towards their objective. This was accomplished by having a secondary Cinemachine brain with a separate camera.

To implement this feature, we adapted our existing smart camera system to not just show off local events but global ones too. This was attached to a second camera which would take input from many sensors (similar to that of the AI) and find the best angle. This would then be broadcast to every player in the lobby.

However, this caused issues due to the additional rendering required, reducing the framerate. One of the key focuses of our game is to optimise play as much as possible, and we deemed the framerate drop unacceptable, so the feature was axed. A low framerate shows off our mesh deformations in less detail.

# References

---

- [1] Tasks by Planner and To Do: <https://docs.microsoft.com/en-us/microsoftteams/manage-tasks-app>
- [2] FREE Skybox Extended Shader by BOXOPHOBIC: <https://assetstore.unity.com/packages/vfx/shaders/free-skybox-extended-shader-107400>
- [3] Realistic Buggy Kit by Mehdi Rabiee: <https://assetstore.unity.com/packages/tools/physics/realistic-buggy-kit-62978>
- [4] “If” font by Chris Hansen <https://www.1001fonts.com/if-font.html>
- [5] “Baron Kuffner” font by Bumbayo Font Fabrik: <https://www.1001fonts.com/baron-kuffner-font.html>
- [6] Lucidchart for Microsoft Teams: <https://www.lucidchart.com/pages/integrations/microsoft-teams>
- [7] GitHub <https://github.com/>
- [8] Autodesk Maya: <https://www.autodesk.com/products/maya/overview>
- [9] Google Cloud Services: <https://cloud.google.com/>
- [10] Cinemachine: <https://unity.com/unity/features/editor/art-and-design/cinemachine>
- [11] PeerJS: <https://peerjs.com/>
- [12] The Coturn TURN Server Project: <https://github.com/coturn/coturn>
- [13] GitKraken: <https://www.gitkraken.com/>
- [14] GitHub Desktop: <https://desktop.github.com/>
- [15] Unity Game Engine: <https://unity.com/>
- [16] WebGL: <https://www.khronos.org/webgl/>
- [17] Unity documentation: <https://docs.unity3d.com/Manual/webgl-memory.html>
- [18] Photon Unity Networking: <https://www.photonengine.com/pun>
- [19] TextMeshPro: <https://docs.unity3d.com/Manual/com.unity.textmeshpro.html>
- [20] Terdiman, D., 2015. And the Oscar for best visual effects goes to...Autodesk’s Maya. VentureBeat. Available at: <https://venturebeat.com/2015/01/15/hollywood-fx-pros-i-want-to-be-an-oscars-maya-winner/> [Accessed 14 May 2021].
- [21] NVIDIA’s Open-Source PhysX Physics Engine: <https://github.com/NVIDIAGameWorks/PhysX>
- [22] Photon Unity Networking: <https://www.photonengine.com/pun>
- [23] Unity forums: <https://forum.unity.com/threads/multithreading-and-webgl.817986/>
- [24] Unity documentation: <https://docs.unity3d.com/Manual/webgl-interactingwithbrowserscripting.html>

- [25] Physics in Unity:  
[https://docs.unity3d.com/2020.1/Documentation/Manual/PhysicsSection.html#:~:text=Built%2Din%203D%20physics%20\(Nvidia,2D%20physics%20\(Box2D%20engine%20integration\)\)](https://docs.unity3d.com/2020.1/Documentation/Manual/PhysicsSection.html#:~:text=Built%2Din%203D%20physics%20(Nvidia,2D%20physics%20(Box2D%20engine%20integration)))
- [26] Unity documentation: <https://docs.unity3d.com/ScriptReference/Collision.html>
- [27] Unity documentation: <https://docs.unity3d.com/ScriptReference/ContactPoint.html>
- [28] PhysX documentation:  
<https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/Simulation.html#split-sim>
- [29] PhysX documentation:  
<https://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXAPI/files/structPxPairFlag.html>
- [30] Unity documentation: <https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html>
- [31] Unity documentation: <https://docs.unity3d.com/Manual/webgl-gettingstarted.html>
- [32] Unity documentation: <https://docs.unity3d.com/Manual/class-Mesh.html>
- [33] TerrainObjExporter: <https://wiki.unity3d.com/index.php/TerrainObjExporter>