University of BRISTOL

COMS30400: Group Project

Big Macintosh

Team Manager: Krzysztof Gora
Lead Programmer: Riley Evans
Jack Bond-Preston
Liam Dalgarno
Adam Deegan
Charlie Haslam
Dimitris Papapostolou

Crazy Parking

Contributions

Krzysztof Gora - 1

Riley Evans - 1

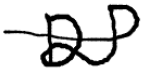
Jack Bond-Preston - 1

Liam Dalgarno - 1

Adam Deegan - 1

Charlie Haslam - 1

Dimitris Papapostolou - 1



Top Ten Team Contributions

1. We designed all assets in the game.
2. We built a tool to allow use to generate roads on the map
3. Built an event-based networking protocol on top of UDP.
4. Created an authoritative server which corrects and synchronises clients. Clients are assumed to travel using the dead reckoning strategy.
5. Created a minimap to view all players in the game.
6. Implemented a driving dynamics system.
7. Built a low-poly shader with Fresnel shading using Unity Shader Graph.
8. Managed the project effectively using GitHub projects.
9. Completed a tool to split the map into chunks, to make designing a larger map possible.
10. Started work on a GPS system to help navigate the players along roads to a parking space.

Categorised Team Contributions

Team Communication

We decided to split into two main development teams, a UX/Game Design team and a Networking/Backend team. We used Slack to communicate as a team. This allowed for easy sharing of resources and to organise our meetings. Regular face-to-face interaction was very important for us; we ensured that during the term we met in person at least twice a week. At these Sprint Retro and Planning meetings we would discuss what we had achieved each week, what perhaps didn't get as much progress as we would have hoped and what we wanted to work on for the upcoming weeks.

We would also take the time to plan new features and decide on the core requirements for these systems. This meant that pairs/teams working on them would have clear objectives and knew what to work on during the next sprint.

Technical Depth & Understanding

Over the course of the project we learned a lot of new skills and strategies for not only game design but software development and working as a team in general. Areas such as networking were a big focus for us. We spent a lot of time comparing various solutions and reading technical articles from both Unity and current game developers on the various solutions they use.

We also spent a lot of time learning about driving dynamics (simplified for game use) and how physics simulation works in a game engine like Unity. Getting to grips with the wheel colliders and balancing realistic simulation with fun gameplay was quite a challenge but we started simple and kept improving our driving systems until we were happy with the way the cars handled.

Throughout the process we realised that there were many intricacies involved. Everything has to fit together in a cohesive way that makes it fun and rewarding for players. Quite a few times we had to completely change the way we thought about a particular game mechanic to make it enjoyable.

Technical Implementations

The network is UDP-based and event-oriented, i.e. the client sends location update events to the server every tick and the server responds with location update events for the other clients. It went through multiple iterations, eventually leading to an authoritative server with a 'dead reckoning' strategy. This implementation requires improvements to make the client see themselves in a smooth, interpolated manner. However it is already much better for the client's view of other clients - these are synchronised in a smooth way and with authoritative server collisions.

Professional Development

Right at the start of the project we made a commitment to try and develop our game in as professional a manner as possible; making use of git and its various features helped with this immensely. We made sure to be using separate feature branches and writing proper pull requests that could be reviewed (requiring at least one approval) before code was added into the master branch. Using GitHub's issue tracking also proved very useful - we were able to combine these with the project boards to create clear objectives and break down our features into key components, making splitting up tasks much easier.

Later into our development we started to use unity cloud build. This enabled the automatic building of our project (which was often time-consuming) when new features were implemented.

Game Playability

Since our game was supposed to be quite fast paced and chaotic, we wanted it to be quite easy to control the cars and play the game. We allowed the use of the arrow keys to control the car but also gave the option of using the WASD keys so that players could use what they were more comfortable with. The game also supports the use of a gamepad to control the car, which could be useful for accessibility in a few cases, or if a particular player has a preference for a gamepad.

At several stages we changed the way the driving handles and the overall 'feel' of the cars to make them easier to drive and more predictable. This improved the player experience a lot and helped the game feel more cohesive.

Look & Feel

We wanted the game to have a distinct low-poly look. This would additionally benefit us as we were targeting the lab machines with low-spec hardware. As a result, all assets (models, textures, terrain) and shading were created entirely by the team.

Using bespoke 'in-house' assets allowed for a fine-grained control over the style of the game. This meant that all the assets were homogeneous, giving the game a more uniform look and feel.

Game Novelty

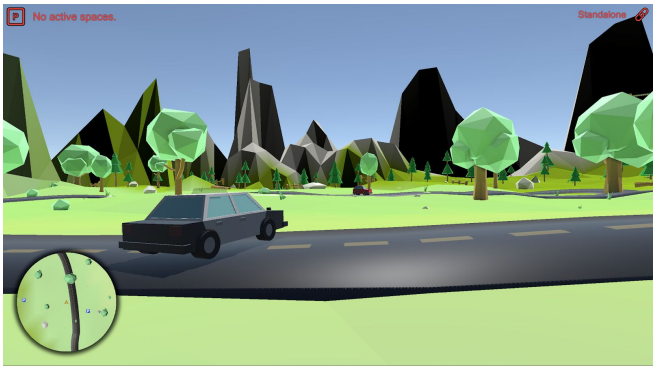
The game would have had collision physics that created a fun experience for people to crash together. This was planned to be done with predefined collision patterns. For example a player could be simulated to barrel roll when they are hit by another car in the side.

Abstract

Crazy Parking is a battle royale style knockout game designed for tens of concurrent players. Each player is tasked with driving around a world to try and find a parking space before a timer runs out. Players can fight each other for a space by colliding and pushing others away. However, once a player has parked in a space it doesn't mean they automatically advance to the next round. They must still work hard to defend their space from other players who will try to steal it from them before the round is over.

After each round, players not in a space are eliminated from the game and brought back in the next round as police cars while the remaining ones have to try to park in a reduced number of spaces. The task of the police is to stop cars from parking in a space by pushing them out the way and hindering them.

The rounds continue, each with fewer participants, until the last player remaining is crowned the winner.



Team Process & Planning

Communication

The group has worked closely together. We split into two main development teams, a UX/Game Design team and a Networking/Backend team. This however was quite flexible, because we were keen to not box people into specific roles. People occasionally moved around and worked on different areas when more work was needed.

Working in person in MVB was always the most effective. We used pair programming extensively throughout the development of our game. When working in pairs, we were able to ensure that bugs were caught early and any uncertainties were resolved early. This was particularly important in this project due to our lack of experience in Game Development, and especially the Unity Framework.

When working from home, it was extremely hard to still carry out work in pairs. We were therefore focused on ensuring that in person meetings happened as often as possible.

Whether working at university or from home, we highlighted the importance of communication at all times. Slack[1] was used for team discussions and organisation. We split it into multiple channels such as '*team-organisation*' and '*proj-networking*' to keep information relevant and easy to find. For remote work, we combined the use of Discord[2] and Visual Studio Live Share[3]: Discord allowed for group video calls and screen sharing, whilst Visual Studio Live Share allowed for real-time collaboration on the same codebase. This approach proved very effective throughout the project.

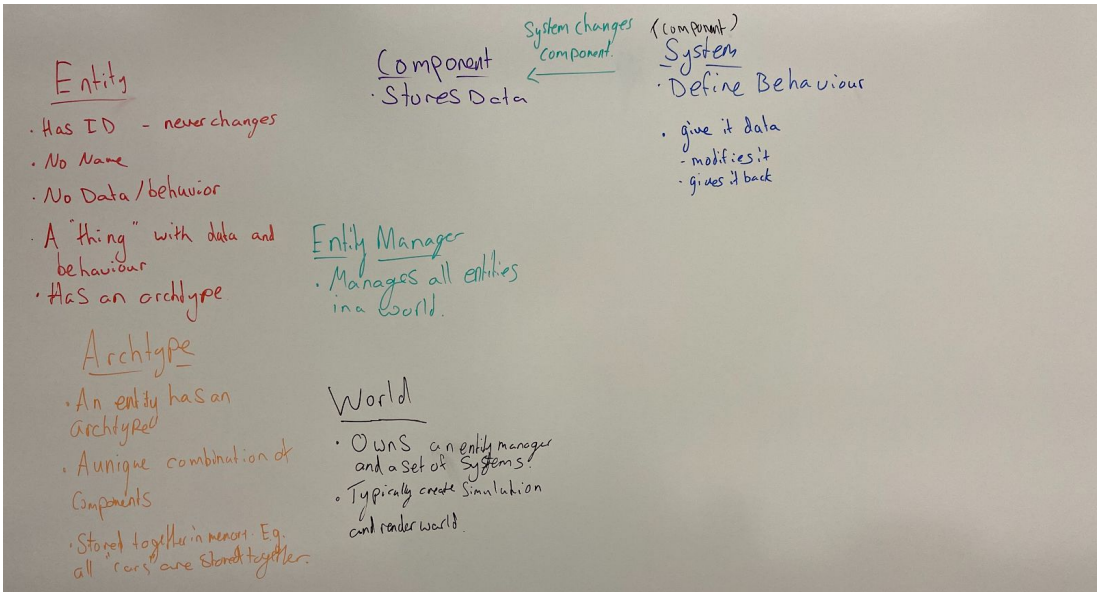
If any problems occurred, we were able to resolve them quickly thanks to the many ways of communication that we introduced. All team members were able to voice and address concerns via Slack, whilst issues and comments related strictly to code were placed on GitHub issues and pull request comments.

Planning

Originally, we didn't know where to start. We settled on using Unity since Charlie already had experience with it and it is cross-platform. The first few weeks of our time was spent experimenting and planning on the technologies required, which proved more difficult than

[1] https://slack.com/
[2] https://discord.com/
[3] https://visualstudio.microsoft.com/services/live-share/

expected. Unity has many competing technologies such as: Entity Component System (ECS) vs. Monobehaviour; Unity NetCode vs. UNet; and Universal Render Pipeline vs. Built-In Render Pipeline. The former are newer and somewhat experimental, so we were unable to take advantage of ECS and Unity NetCode. We settled on using Monobehaviours, our own networking solution, and the Universal Render Pipeline.



A discussion of ECS on a whiteboard from a team meeting.

Once we had decided on the technology stack to use, we split into teams and began planning the required features on GitHub. Initially we planned informally through meetings and Slack, which led to confusion over the end product. We identified this problem and therefore had a meeting to make final decisions on the required features in the final game. As a result, we kept documentation under the GitHub Wiki which outlined the coding style, intended feature list, and the feedback received on presentation days.

We kept quite closely to the main idea of the project - a car-based battle royale. The more nuanced features, however, went through many iterations as a result of play testing and feedback. We felt that it was more important to remain responsive to feedback and improve our game iteratively than to stick to a strictly predefined plan.

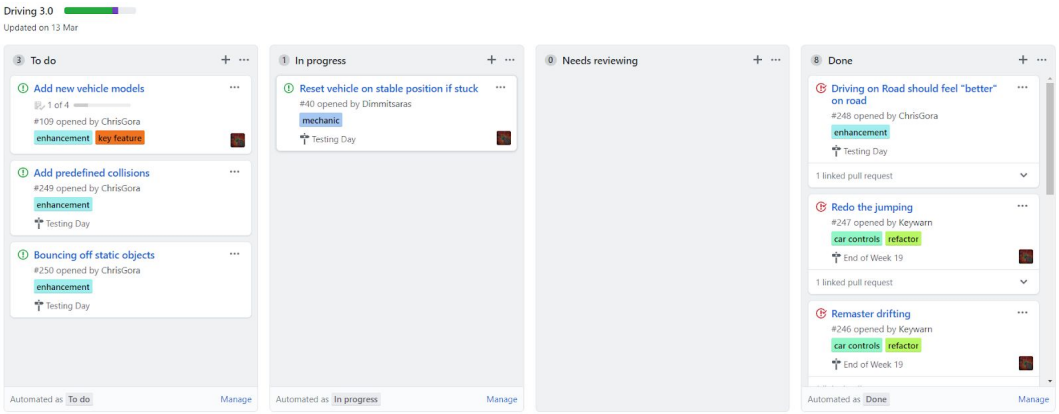
Some parts of the game were undefined until the previously mentioned meeting. In retrospect, it would make sense to have agreed on the features and scope earlier on. However, at the beginning of the project we found it very difficult to judge difficulty and time needed for various features due to our lack of experience with Game Development and the Unity Framework. As we progressed in the project, we became much better at knowing how much work can be done and what is technically viable.

Project Management

We used Github as our project management tool, taking advantage of the Kanban-style project boards, pull requests, branches, etc. It was agreed that commits directly to the master branch should not be allowed and each pull request had to be reviewed by at least 1 other team member in order to be merged in - ideally two who were familiar with similar features in the project. As a result we could ensure that everyone knew what was being done and that they were happy with the quality of the work. Constructive feedback in the reviews was common and encouraged.

Features were often discussed in meetings or Slack, then added to the GitHub issues board, before finally being worked on. Using issues allowed us to clearly identify what features need implementing, who is responsible, tag them according to project/theme, and automatically close them when a matching pull request is merged. We used milestones to organise tasks with time deadlines so that everyone knew which tasks needed to be completed sooner. We could then monitor our progress towards more significant deadlines, such as the playable prototype review.

Issues were used to represent an ‘atomic’ feature that required attention, which were then grouped together using the Github project boards. Using project boards allowed us to track the progress of an issue from submission all the way to the merge stage.



Example usage of a GitHub project board showing the ‘Driving 3.0’ project.

Equipment

It was extremely tedious to test our project. Since it is multiplayer, it requires a server and 1-2 clients to test the network sufficiently. Unfortunately, multiple instances of a single Unity project are not permitted, which forced us to either: build the entire server and client, or duplicate the project twice and run up to three separate instances of Unity. This proved impossible on some team member’s personal machines, since each instance of Unity can use around 1-2GB RAM. This was mitigated at university by using the Apple Mac Pros that were available in MVB 3.43, which served as our development machines throughout the project.

Individual Contributions

Krzysztof Gora

Local Network Testing in MVB 2.11

At the beginning of the project, developed a simple Golang server to stress test the MVB 2.11 Local Area Network. The promising results influenced the design process for the Game Network architecture later in the project. It also allowed us to set realistic expectations in terms of concurrent player numbers.

Design and Implementation of a scalable Networking model

Through the project, the network was designed and improved in an iterative manner. It started with basic coordinate sending and evolved into a highly sophisticated framework supporting synchronisation of a large number of concurrent players.

Development of the Parking Space logic

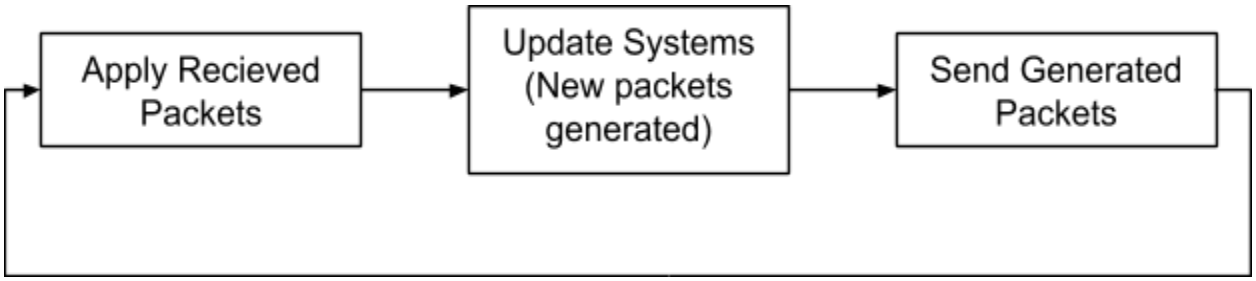
Developed all of the client and server code required to correctly support Parking Spaces. This required careful planning and design to ensure that the state is synchronised between all clients and the server. Synchronisation and consistency were essential when developing this key game mechanic.

Testing Framework for the Networking Backend

Later in the project, refactored large backend classes to introduce better separation between Unity MonoBehaviours and actual Game Logic. The new Game Logic never directly interacted with any Unity APIs. This in turn made it much easier to implement Unit Testing for the backend logic classes, speeding up development and bringing the many advantages of Test Driven Development. Prior to this change testing the game was manual and required multiple instances of Unity just to start the game.

Riley Evans

Network Control Loop



A game loop was created to allow for linking and management of all the systems in the game that need to have some aspect of network control.

C# Networked Event System

A system was developed that would allow for C# events to be sent easily between client and server. Systems are able to subscribe to different events in the system, they can then be informed when they occur. These can also be broadcast across the network between the server and client. For example a round start event can be triggered on the server, this can be broadcast to all systems in the server as well as all clients.

This makes it easier to add new systems to the server and client, they are able to subscribe to the events without having to modify any of the current systems.

SatNav System

Started to work on a SatNav style system to allow people to easily navigate themselves to a parking space. At the start of the game it would analyse all the roads to find intersections. It was then planned to join these up to form a graph network of roads. Players would then receive turn-by-turn directions to one of the parking spaces on the map. This could have been exploited to cause more players to meet on the map by navigating them on similar routes.

Unity Cloud Build

Setup the project to automatically be built when any changes are pushed to master. This ensured that we always have a compiled binary for the master branch which could be used for testing at any point.

Project Management

Planned next major steps in development, assigned people to tasks that needed to be completed.

Jack Bond-Preston

Networking

Helped research and implement multiple iterations of the network, including original abstractions and the final WIP improved network. Participated and contributed to many discussions throughout development on how to proceed with the networking. The original network simply sent all the player positions and forces at a high tickrate to all other players. This worked okay and was playable, but used too much bandwidth to be scalable and had issues with jitter. Also helped with getting the local multiple Unity instance setup working to develop the network.

Implemented network event queueing and dequeuing, and the system to pack multiple events into one packet. This allowed us to send network packets only in FixedUpdate in the Server class, as opposed to whenever they happened to be generated.

Worked on a lot of the logic for sending positional and force data over the network, in both the original and upgraded networking setup. Implemented almost all of the original syncing of player positions to create the basic, uninterpolated multiplayer.

Parking Space Logic

Implemented system for selecting parking spaces at the start of each round, as well as assisting with other areas of parking logic code.

HUD

Redesigned and improved the in-game heads up display, including creating vector graphics and rewriting a lot of the internal logic to allow it to display new information that would be useful to the players (such as more information about available parking spaces, when your space is taken, etc.).

Pull Requests

Assisted with code reviews on Github pull requests to ensure a high standard of programming. Participated in most group discussions about the directions of the game and how to approach various elements of the game and programming.

Liam Dalgarno

C# and Code Reviews

As the team member with the most C# experience, I assisted with many features throughout development, specifically when setting up the project and getting the network started. Took part in many pull request code reviews and fixed bugs, especially regarding the low-level backend code and C# specific features such as delegates and events.

Networking

Most of my time was spent working on and researching the network. Built the first iteration of the network which was oriented around event-based packets as outlined in the technical content section. I was involved with implementing the (de)serialisation of data over the network, and the eventual abstraction into bespoke event-based packets.

The improved version of the network required extensive research since we knew very little about real-time networking. I was responsible for this; I pulled together multiple sources[4] [5] [6] documenting online multiplayer and decided on a server authoritative model with dead reckoning. I was also primarily responsible, assisted by Jack and Chris, for the implementation of this new network. Although it was unfinished, we made significant ground and improved the consistency of the network massively.

Minimap System

Integrated the game state information into an easy to use minimap script. Initially, this involved rendering the game twice: once from the perspective of the map, and once from the player's perspective. This had a massive performance hit due to the double rendering of the game, so I eventually switched it to a more specialised 2D solution. The improved version involves a main high-resolution map texture, which is scrolled relative to the player's position and rotation. The map texture is effectively a static image taken from a birds eye view. Then, any GameObject can be rendered to the minimap by simply attaching the `RenderToMinimap` script and linking it with a texture.

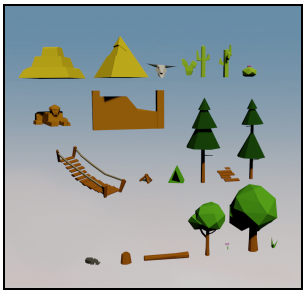
[4] https://www.gabrielgambetta.com/client-server-game-architecture.html
[5] https://gafferongames.com/categories/networked-physics/
[6] https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

Adam Deegan

I mostly worked on the art and map creation.

Models

Initially the map was going to be one big city, but I decided to split it into parts with a distinct theme: the player should be able to easily recognise where they are on the map. It also allowed for more variety in scenery.



Afterwards, I came up with a list of around 50 models grouped by the area of the map and started to create them using Blender. I had to learn how to use Blender[7] as I hadn't before - as such, the design of the models and map were kept 'low poly' to decrease the complexity of models, and thus the time it took to make them. Colours were then added by using materials in Unity, so a consistent styling could be kept.

Terrain Management

After deciding we wanted a low poly terrain, I started to look into ways of achieving this. The standard Unity Terrain tool didn't (easily) support low poly terrain, so I set out to create our own tool. This involved researching how to generate Mesh objects in Unity and manipulating them at runtime, as well as an Editor tool, however it quickly became apparent that the scope of creating the tool would be too large.

Instead, I looked towards solutions already available and opted to use Polybrush[8]. This turned out to be a bit weird to install and use since the documentation was outdated. There was still an issue however; the automatic 'chunking' of the terrain that Unity's Terrain object did (which would help with optimization) would not work on a singular object manipulated by polybrush. I made a terrain chunking tool, Chunkster, to take care of the management of chunks and the 'stitching' together of them.

Shader

The default shader that came with Polybrush used smooth shading, but the low poly look required flat shading. I researched creating shaders, for which I had to understand lighting models in video games, and made a prototype by editing said shader that came with Polybrush. Alongside this I added some extra lighting features such as Fresnel.

After our project's rendering pipeline was upgraded, I had to switch the shader over from code to ShaderGraph[9], Unity's graphical shader editor. This took some time to understand but resulted in a format which was easier to understand and edit.

[7] https://www.blender.org/
[8] https://unity3d.com/unity/features/worldbuilding/polybrush
[9] https://unity.com/shader-graph

Charlie Haslam

Unity Experience

Since I was the only person on the team with (some) experience using unity, I worked on the front-end of the project. I worked on various systems and helped some of the others on the team to get started with their areas of the project and showed them some of the tips and shortcuts I had learned previously.

Road Tool

Unity enables users to create their own extensions of the editor, enabling a faster work-flow and easier development. Since our game revolved around driving and roads I thought it would be useful for it to be as easy as possible to create these roads. I created an editor tool that allows for the generation of road models using a spline[10]/curve system (a bit like the path tool in an image editor for example). The system then automatically creates a road using this curve. This meant that we didn't need to make new models for each section of road and made it much easier to design our levels. I then improved this so that the road would be altered to fit the shape of the terrain beneath it, making it even easier to design roads that fit with the level.

Driving Systems

Initially we used wheel colliders[11] to create the driving mechanics but we were having issues with the stability of the car. I then decided to create a new solution essentially from scratch without using the built-in wheel colliders unity offers. This resulted in a slightly better experience but wasn't very easy to control and didn't feel very satisfying when tested in the game. We switched back to wheel colliders and I spent a lot of time trying to learn how they were simulating the physics of wheels to help tweak the settings for our car and how it behaved. In the end, the new driving system was a huge improvement over previous iterations.

Modelling Cars

Since I did most of the work on the car mechanics, I did the modelling of the vehicles in the game. I tried to match Adam's 'low poly' style and came up with a car model we liked and behaved well in-game.

Collision Physics

We wanted our players to interact in the game by driving into each other and creating these crazy collisions with exaggerated physics. To start we created a system which would let one player 'win' a collision based on who had the highest momentum and then pushing the other player away. This idea worked but didn't feel great when we tried it in demos. We later came up with the idea of having some pre-simulated collision animations to make the game feel more arcade-like.

[10] https://en.wikipedia.org/wiki/Spline_(mathematics)
[11] https://docs.unity3d.com/Manual/class-WheelCollider.html

Dimitris Papapostolou

UI/UX functionality

Implemented basic game and UX/UI features such as a main menu working as a launcher to a game session, HUD elements displaying important round events or information like the vehicle's speed, and settings that can be launched and changed through both the main menu and an escape key pop-up menu.

A notable feature to be implemented was a colour wheel selection tool for picking a car colour, in addition to a default scroll of basic colours.

Camera

Created a basic camera script to follow the player's vehicle. A large improvement over a typical camera script is the ability to maintain a horizontal view without tilt when the vehicle is climbing terrain or just flipping around with the arcade-style jumpy physics of the game. This avoids getting the camera below terrain, making it very unpleasant for the player.

Software

To build the game there is a script called build.sh, which creates a directory containing the client and server binaries. The server can then be run using `./server -nographics -batchmode`. The client can be run by just double clicking the `client` executable file.

To play the game with two players you will need to run a third client, we refer to this as the admin client. It is used to control the server since the server is a command line application that cannot take any input. Once the server is running and you have obtained its IP address, you can start the 3 client applications. Then for each client click “Play”. For the two playing clients fill in all the options, then click “Play”. For the admin client input the IP address then click “Back”, then click “Admin”. Once all clients have connected then click “Start Game”, the game should then start.

New functionality can be added to the game, in a relatively simple way. If new objects require location synchronisation between the all clients then they can be created through the World class which controls the objects that require this feature. If a feature does not require location synchronisation, instead just a form of event based communication then it can easily link into the event system and trigger events that will be broadcast to other clients/the server as and when it is needed.

Technical Content

Network

Architecture

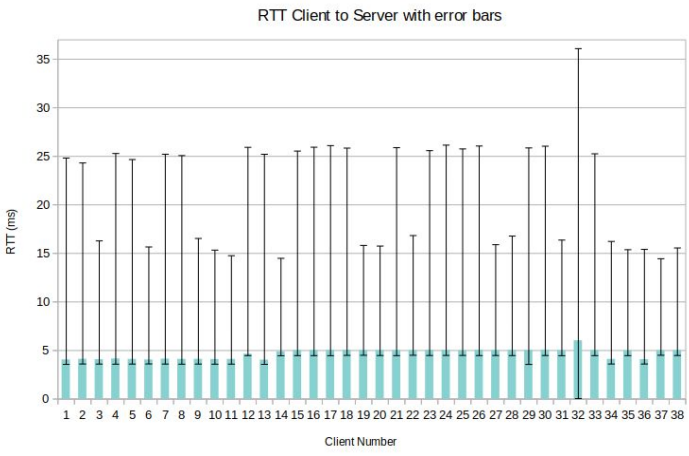
We were faced with a choice between different networking technologies: UNet, Unity NetCode, or create our own. Unfortunately, UNet[12] is deprecated and uses a peer-to-peer relaying strategy, which is not ideal for games like ours that aim to support a large number of players. On the other hand, Unity NetCode[13] is dedicated server model which requires the use of ECS[14]. These are both very immature packages with little documentation and, at the moment, are quite unintuitive. Therefore, we instead decided to create our own networking solution.

Although we decided not to use Unity NetCode, we did use the underlying Unity Transport Layer. This is a thin library which allows for primitive networking operations such as connection over UDP and interaction with network streams.

When deciding on a host we had several options open to us. We could use a cloud provider or have the server hosted by the University of Bristol IT Services. We evaluated both approaches that we had, both had benefits and drawbacks. The server hosted onsite by IT Services would likely have a lower round-trip-time, however questions were raised over how reliable it would be. We made the decision to test out a cloud provider (Amazon Web Services), if the round trip time was reliable enough then we would use the cloud provider.

LAN UDP Stress testing

One of the first tasks we carried out was to stress test the new network system in place in the MVB 2.11 Computer Laboratory. To do this we implemented a simple server in Golang being run in the eu-west-2 (London) availability zone on Amazon Web Services. This would then receive packets from Python clients being run on the lab machines. This will allow us to measure certain metrics about the network, for example we are able to count the number of lost packets and calculate the round-trip-time (RTT). From this we are able to determine how reliable



[12] https://docs.unity3d.com/Manual/UNet.html
[13] https://docs.unity3d.com/Packages/com.unity.netcode@0.0
[14] https://docs.unity3d.com/Packages/com.unity.entities@0.10

the network is, the game's network systems can then be designed knowing what problems we may have to overcome.

In our test we used a t2.micro instance along with 38 clients each being run on a different lab machine. We recorded 5 different metrics in each client: min/max/avg RTT, the number of pings and the number of lost ping packets. A packet was determined lost if a response was not received from the server within 2 seconds.

We are able to have a high confidence in our network architecture. The highest ping observed was 36ms with an average of 5ms: this is acceptable for most network games. The results also showed that of the 1.3 million packets sent only 695 were classified as lost giving us a packet success rate of 99.95%. Although this is high, we do still need to develop a system that is able to cope when a small percentage of the packets are lost while being transmitted.

Network Synchronisation

Data is exchanged through small 'packets' which represent something in the game, such as a player's position, the current round, or the player entering a space. These packets have a 1-byte header which tells the recipient *what* kind of packet it is, so that it can be deserialised and handled accordingly.

The first 29 bytes of a player position packet. The header byte maps to the corresponding class so that the rest of the packet can be correctly deserialised.

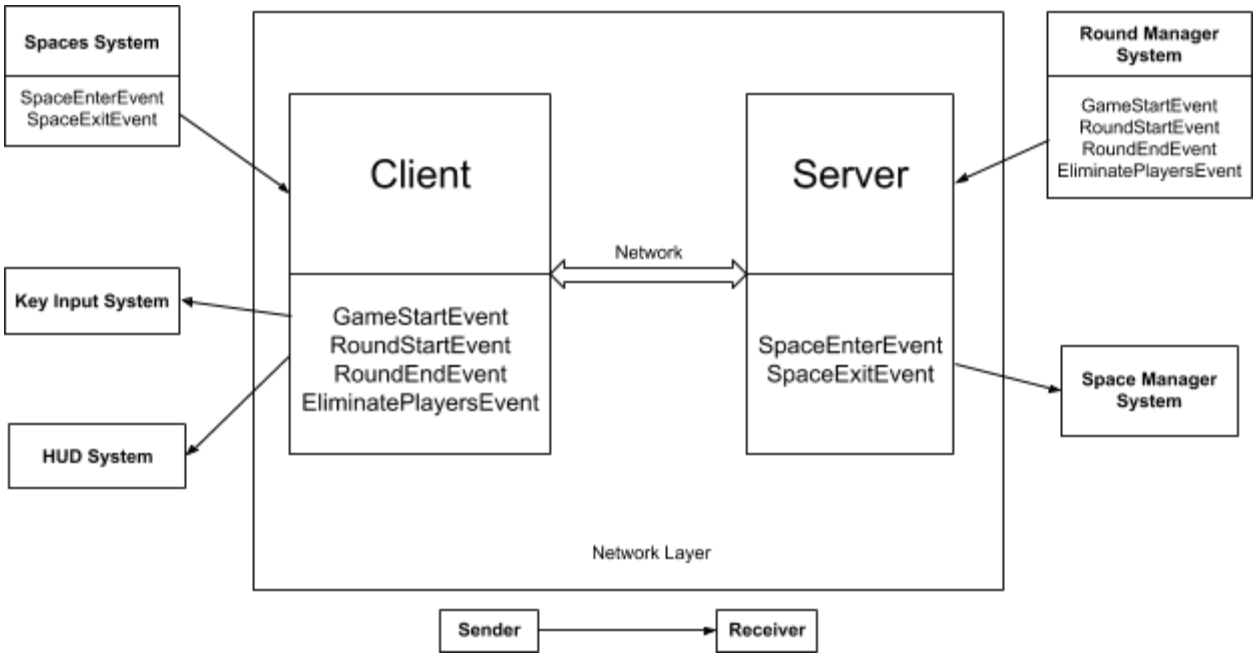
There are two different types of data which must be exchanged: events and player data. Events are one-off and may happen at any given time; they represent changes in the game state such as entering a parking space or start of round. Player data on the other hand is exchanged continuously, since each client needs to know the most recent version of all other clients.

In its current state, the network uses the ReliableSequencedPipeline option, which ensures delivery of packets in the correct order. Although this is still built upon UDP, this has some overhead which can be avoided in some scenarios. The two types of data differ in that event data *must* be received or else the game state will become desynchronised, whilst the occasional player data can be safely missed due to the high update frequency. In the future, the network could be developed in a way such that delivery of event packets is ensured, but player data is not to ensure faster delivery.

Event Synchronisation

To enable us to synchronise events between client and server, we built a system that would use C# events. These events would be mirrored between client and server, this allowed the network layer to be abstracted out from the game logic. To all the systems that are part of game logic, the network layer has been abstracted away - all they are aware is that if they trigger an event that

is synchronised over the network then all the systems on the server will also receive this. The figure below shows the structure of these systems and some of the events that are synchronised.



Player Location Synchronisation

The network is a client-server architecture, where the clients data is trusted by the server at all times. On every client tick, the client will send its current transform data (i.e. position, rotation, velocity, angular velocity) to the server. Similarly, on every server tick, the server will send the current state of the world to all clients.

This approach only works under the assumption that:

- All players have low ping and packet loss.
- The world simulation is deterministic.
- Players act reasonably, i.e. do not cheat.
- Player interaction is infrequent.
- The tick rate is high enough to give clients the illusion that the other clients are moving smoothly.

Clearly, this system is not viable for a fast-paced battle royale: player interaction is strongly encouraged and Unity physics is certainly not deterministic. We transitioned to a different server model whereby the server is authoritative over the state of the simulation. Rather than the clients sending their state to the server, they instead send their desired inputs. This way, the server will simulate the result *for* each client and will have the 'final say' over the state.

This does have some drawbacks, however. The clients will need to wait for the server to simulate their inputs before they get feedback, which leads to disjointed and jerky gameplay. This can be remedied by making the client also simulate the state, and then be retroactively corrected by the server if their state differs too much from the true state. Additionally, to

smooth out the behaviour of others, a client can assume that the inputs of other clients persist from the last update, i.e. they assume that the player is still holding the same keys they were on the last update from the server. This strategy is known as 'dead reckoning'. This is a valid assumption to make, since cars cannot turn or accelerate instantaneously, and with a frequent enough update rate from the server, it gives extremely accurate simulations.

Unfortunately, we were not able to finish this transition. We were able to implement a server with dead reckoning, which resulted in very clean simulations. However, the client's view of themselves had jitter. We believe this is caused by network jitter - the tendency of packets to cluster together rather than be a constant stream - and could be fixed with a jitter buffer and some interpolation between updates.

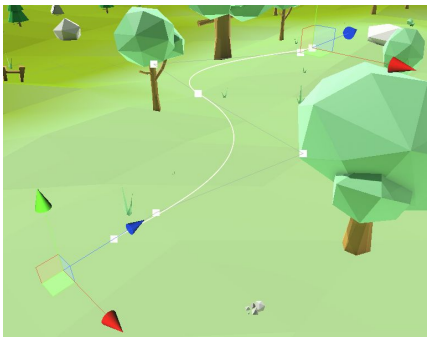
Our solution also required some optimisation if it were to support upwards of 50 players. The state is sent in its entirety on every server tick, which scales strongly with the number of players. With more time, we planned to implement delta compression: an optimisation where the server sends the *changes* since the last acknowledged packet rather than the entire new state. Furthermore, we planned to selectively update each client dependent on their position, since a client only needs to know about clients within their render distance.

Road Tool

Requirements

To aid in development we wanted to create an editor tool to help us easily create roads in our levels. This required a fair bit of thought and consideration to decide how we were going to implement this. We did some research into creating plugins and editor scripts for Unity, deciding this would be an appropriate solution.

Curves



We ended up creating some classes which represent certain kinds of curves (bézier[15] for example) and a spline (collection of polynomials/curves). We then made some editor scripts which hook into the Unity engine to enable visual and immediate editing of these curves from within a scene. Meaning that once the system was created, curves and eventually roads could be defined from a completely visual interface, just like moving objects around in a scene.

Paving the Road

Once the curve is defined, the plugin allows a user to define the desired width and depth of their road. They can also specify a texture to use on the created mesh. Creating these meshes involved a lot of 3D geometry and mathematical knowledge.

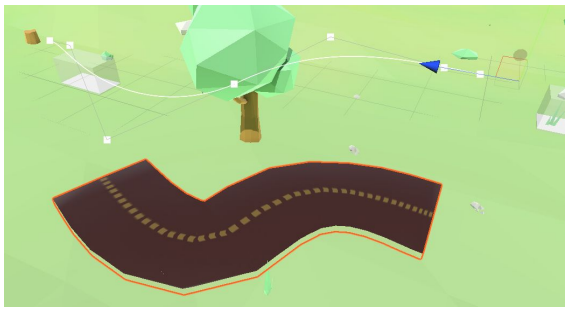
[15] https://en.wikipedia.org/wiki/Bézier_curve

Originally, we designed it so that the plugin would create the roads as the game loaded. This worked for our small test scene but eventually it proved quite cumbersome and had a negative impact on load times. We decided it would be better to generate these roads in-editor. In this new version, there is a button in the inspector panel (under the boxes for user defined width and depth) that when pressed, creates the mesh for the road. This mesh can then be treated just like any other object in the game. It can be repositioned in the scene and even saved as a prefab for use later, without a need to use the original curves. If the curve needed to be changed, simply pressing the button again would generate a new updated road mesh.

Snapping to Terrain

Initially, the roads just followed the curve but we realised this made it hard to fit the road to the terrain in our levels. Where the terrain was uneven, more points needed to be added to the curve to try and fit it as best as possible to the terrain. This was quite a painstaking process and still resulted in the road mesh occasionally clipping through the floor or being too far off the ground. Resulting in roads that were hard to drive on.

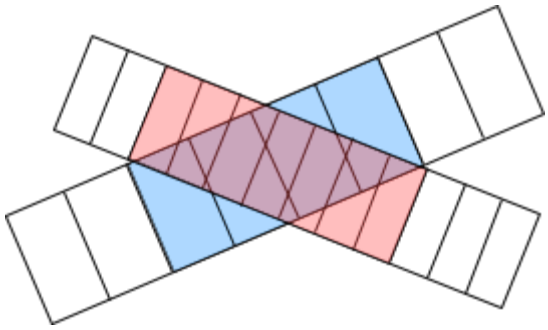
We used unity's in-built raycasting to solve this issue. When generating the road, we first calculate where the left and right side of the road should be based on the curve. These are then split into a number of user-defined segments. Each segment casts a ray down to see how far the terrain is from the road. The road is then moved to sit just above the terrain (with a user-defined parameter). This allows the road to undulate and follow uneven terrain. Resulting in a much better aesthetic and an easier driving experience.



A generated section of road which has been fitted to the underlying terrain

Joining Roads

One thing that we needed to do once the roads had been generated was to be able to detect where they overlap, this would have allowed us to overlay junctions and add support to the SatNav for turn by turn navigation. This poses some challenges though - it was not possible to easily detect if two roads overlapped. It was possible to get bounding boxes from two road GameObjects but since the roads allowed curves the boxes may intersect even if the roads do not. We devised an algorithm that would create bounding boxes for smaller chunks of the road, then work along each road detecting sequential chunks that overlap with another road. These bounding boxes could then be joined together to form a bounding box that encompassed the whole junction. This can be seen in the following example. A bounding box would be formed that encompasses the areas that have been shaded red and blue.



Chunk Tool

Description

After deciding to use Polybrush for terrain manipulation (vs Unity's Terrain tool or building a terrain in Blender), we needed a way to split up the terrain for optimization reasons. Polybrush only works on one `GameObject` - if we built all of our terrain on one `GameObject` we would not be able to benefit from Occlusion Culling[16]. This was built as an editor script for Unity, much like the Road Tool above.

Storing Chunks

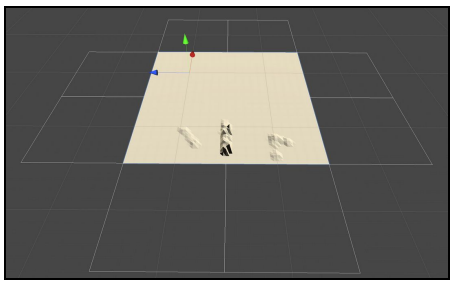
There is a 'parent' container which stores all the chunks via a script, which themselves are `GameObjects`, and has access to them. The parent is given a reference `GameObject` which it clones for any further chunks added - this will only work for a plane.

The parent was initially going to have a dictionary of all the available chunks and access them by their coordinates. There were a few issues with this: dictionaries were not serializable with Unity's serialization so a separate serializable dictionary class would have been required; and consistency errors in the dictionary would be hard to address (i.e. if a chunk were to be deleted it would need to be reflected in both the dictionary and individual `GameObject`). These were both extra complications which would take up more development time.

Instead, we opted to simply access Chunks by name of the GameObject. Whilst this is significantly faster as the number of chunks increases (accessing `GameObjects` by name requires a for loop over all the children of the parent), the map was not going to be massive so the number of chunks would not be either.

Manipulating Chunks

Inside the editor, you can add and remove Chunks by selecting the 'parent' `GameObject` and clicking on the appropriate button in the Inspector panel. A series of empty planes will appear which you can click on to do the appropriate action.

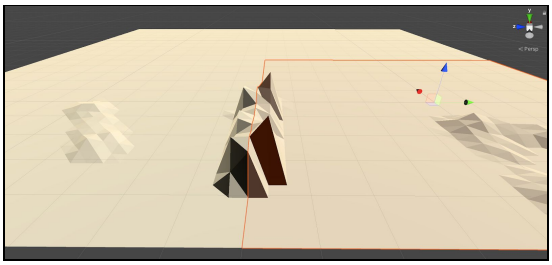


Stitching Chunks

Unfortunately, Polybrush could only manipulate the mesh of one `GameObject` at a time. If there were any inconsistencies between the edges of the Chunks it would look disjointed - for example, a shared edge between two chunks could have a hill on one chunk and a flat section of ground on the other.

[16] https://docs.unity3d.com/Manual/OcclusionCulling.html

To fix this, we needed to be able to combine or 'stitch' Chunk edges together. The high-level (unoptimized) algorithm behind this is fairly easy; for each Chunk, find its edges and stitch them with the adjacent Chunk's edges.



In practice, finding the Chunk edges was hard. Given that a Mesh is essentially represented by an array of vertices/triangles with no guaranteed order, how could we define what vertices are contained in a given Chunk edge? Without manually defining them per model there is no trivial algorithm. Additionally, making sure the correct mesh was manipulated at 'runtime' (Polybrush's mesh vs. the cloned mesh) and that the edges contained the most recent vertex value was difficult.

The approach we took was very constrained and only worked on meshes which are planes. Given a list of triangles (which are comprised of three vertices), you can work out which vertices are either a) on an edge, as they are used in only two triangles; b) on a corner, as they are used in one or two triangles; c) in the centre of the mesh, as they are used in more than two triangles. Once you have identified all the vertices involved in edges, you can figure out which edge they are on: left, right, top, or bottom. This is done by finding the max and min values for the x and z coordinates; identifying the vertices with the appropriate max/min values; and grouping them together.

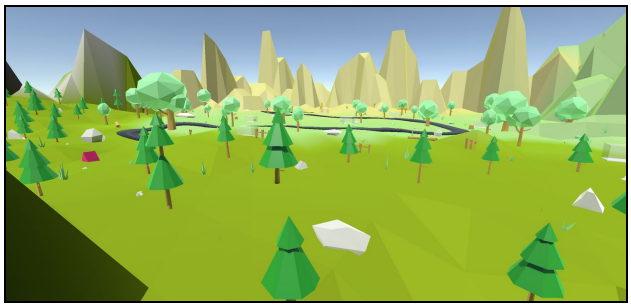
Once this is done, you can apply the high-level algorithm described above between the correct pairs of edges - the edges will have to be sorted when doing this to make sure all the vertex indices line up correctly. You can apply any algorithm for combining two sets of vertices together you want. We averaged the y (height) value for all adjacent vertices.

Stitching all chunks together is done by clicking a button in the same Inspector panel as adding and removing chunks. It is done all at once to mitigate lag during editing the mesh.

In retrospect, this could have been solved in a neater way by generating the reference Chunk `GameObjects` (which are cloned at runtime) and storing a reference to its edges. However, you could argue that our method is slightly more adaptable towards different models.

Shader

Description



The default shader that came with Polybrush used smooth shading, however we wanted flat shading for the terrain. This required a custom shader which needed to work alongside Polybrush, taking in colour information.

Implementation

The shader that came with Polybrush was edited to use per-vertex lighting rather than per-fragment, achieving a low-poly look. This then had to be translated over and created from scratch in Unity's Shader Graph. As a whole this was challenging since none of us had experience with writing shaders before. A lot of settings were changed around until the result was nice, such as adding Fresnel and adjusting the impact of light/material colour.