Team B - NPC Multi-storey Car Park

Game Name: Elemental Explorers

A Virtual Reality Experience in a Procedurally Generated World

University of BRISTOL

Team Members

Matthew Swann - Team Manager
George Wigley - Lead Designer
James Millan - Lead Programmer
Stephen Brock
Imran Zamin Ali
Alex Elwood

Top Five Contributions

1. We planned and implemented a procedural pipeline tool that was used to procedurally generate real-world environments from GIS data.

2. Implemented a plethora of optimisations including GPU instancing, GPU occlusion/frustum culling, merging meshes by material, integration of InstantOC dynamic culling and other methods to reduce overdraw or the number of batches to maximise frame rate on the VR hardware.

3. Using NGO, we networked the game allowing two players to connect peer-to-peer while maintaining a high degree of robustness. This included networking scene loads, dynamic spawning of objects, player teleports, and tracking of players' body movements.

4. We created a tactile and kinetic movement engine that utilises virtual reality hardware to create an immersive gameplay experience.

5. We employed an agile development process, with over 1,500 commits, 120 pull requests merged, and closing over 250 tracked issues as a result of 45 team meetings.

Video link:- `https://youtu.be/2H-Qpqbo92M`

Nine Aspects

Team Process

- Weekly agile sprints with pair programming on larger tasks. (see Team Process section)
- Regular meetings commencing on Monday to plan the week, including whiteboarding sessions. (see Team Process section)
- In-person teamwork was a big focus. There is no replacement for it. (see Team Process section)
- Prioritising asking questions regularly to maximize productivity. (see Team Process section)
- Our team would follow up on tasks that have taken longer than expected. (see Team Process section)
- Provide a mixture of tasks people want to do and tasks necessary for our game to progress. (see Team Process section)
- Iterative development led to fundamental changes to how the game would be played. (see Team Process section)
- Extensive planning sessions. (see Team Process section)
- Entire team involved in integration. (see Team Process section)
- Used GitHub's Kanban board, replaced by Monday.com for project management. (see Team Process section)
- Consistently tried to improve our team efficiency by reflecting on the team process. (see Team Process section)

Technical Understanding

- Researched and compiled Gen Nishida's Photo2Building for potential use in modelling famous buildings. Decided it was not feasible to use this for our project. (see Individual Contributions section)
- Researched two different methods for destruction on building meshes. (see Individual Contributions section)
- Research the differences between Photon and Netcode for GameObjects. (see Individual Contributions section)
- Researched Context-Free Grammars and Subdivision with regards to procedural building mass generation. (see Building Mass Generation section)
- Researched the Overpass API and how to read relevant data and apply it to building classes. (see Building Reconstruction section)
- Researched a variety of rendering techniques for instancing. (see Individual Contributions section)
- Researched general optimization techniques to reduce the number of batches and the amount of geometry on-screen. (see Individual Contributions section)
- Researched shaders for use of effects and computation. (see Individual Contributions section)
- Researched how to triangulate procedurally generated meshes and implemented a research paper. (see Building Reconstruction section)
- Researched how to procedurally generate a triangular prism and implemented this using a rectilinear convex hull algorithm. (see Building Mass Generation section)

Flagship Technologies Delivered

Virtual Reality

- Created a movement engine in which players physically pull themselves through the environment. (see Demo / Video)
- Implemented joystick movement and haptic feedback. (see Demo)
- Implemented point-and-click teleportation and a vignette option as methods of reducing motion sickness. (see Demo)
- Allow the user to interact with the UI using the VR controllers as laser pointers. (see Demo)
- Implemented hand position, body position, and height tracking. (see Demo)
- Utilized VR to create an immersive environment. (see Demo / Video)

Proceduralism

- Created a terrain with height corresponding to that of the real world. (see Terrain Generation)
- Included rivers, lakes, and streams on this terrain that correspond to real-world water sources. (see Terrain Generation)
- Reconstructed a road network from real-world data. (see Roads Reconstruction)
- Reconstructed building meshes from real-world data. (see Building Reconstruction)
- Created procedural building mass generation using generators in Blender. (see Building Mass Generation)
- Scattered nature assets on buildings. (see Asset Scattering)
- Optimised grass. (see Optimisations)
- Applied moss textures to the building materials. (see VFX and Shaders)

Implementation and Software

- Created a procedural pipeline node editor that takes the form of a directed acyclic graph. This is able to run in the Unity runtime without causing major frame rate issues. (See Pipeline)
- Procedurally reconstructed building meshes from GIS data and triangulated them (see Building Reconstruction)
- Textured and added building decorations (including roofs, windows, etc) to these buildings procedurally to create photo-realistic geometry for our building mass generation. (see Building Mass Generation)
- Reconstructed terrain with the correct heights and with all water features. The grass was mapped to this terrain. (see Terrain Generation)

- Reconstructed the road network of a given area by converting the nodes to a graph network and implementing a merging algorithm. (see Roads Reconstruction).
- Implemented Dijkstra search to find points of interest and generate a route for the gameplay. (see Individual Contributions).
- Created an immersive movement engine from scratch, having to remove elements of real physics to prevent motion sickness. (see Movement Engine)
- Reworked networking code to improve robustness and long-term bug resolution (see Software Maintenance).

Tools, Development, and Testing

- Used continuous integration and a linter on the project. (see Development tools)
- Used an agile git-flow development style [1], with weekly agile sprints and splitting into agile scrum teams where appropriate. (see Development Process)
- Multiple reviewers on pull requests that tested changes locally before merging. (see Development Process)
- Documentation and good software design reduce the learning curve of understanding the code base. This allows seamless contribution from a new contributor within the team. (see Further Contribution)
- Maintained software by adapting it to fit our needs and switching frameworks if deemed necessary. (see Software Maintenance)
- Produced software tools that made development more efficient for our team. (see Software Produced)
- Creation of several internal tools used for testing and developing new features. The main example of this is the procedural pipeline framework, smaller examples are editor tools created in Unity. (See Software, Tools and Development)
- Frequent user testing in both a moderated and unmoderated environment to guide development. (see User Testing)

Game Playability

- Movement is robust, engaging, and immersive. (see User Feedback)
- Gamemode is balanced to allow users to explore the environment whilst having a competitive element. (see Demo / Video)
- Controls are physically intuitive and make heavy use of VR mechanics. (see Movement Engine)
- Different types of gameplay avoid performing repetitive actions for an extended period of time. (see Demo)
- Minigame style of gameplay gives the user ”breaks” from the movement system aiding motion sickness prevention. (see User Testing)
- Game supports voice chat increasing player interaction. (see Voice Chat)
- Implemented a Vignette to reduce motion sickness. (see Vignette)

Look and Feel

- Created grass that sways in the wind. (see Grass and Asset Scattering)
- Created custom water and fog shaders to improve the environment. (see Technical Art)
- Created our own Foley sound effects and implemented them. These were all of the same quality using similar filters which avoids the jarring effect of varying quality sound effects. (see Demo)
- Included voice lines to guide the user. (see Demo)
- Reconstructed buildings from a chosen area. (see Demo and Building Mass Generation)
- Reconstructed roads from a chosen area. (see Demo and Roads Reconstruction)
- Graphics have a consistent level of detail and theme. We avoided using too high quality or cartoon-style assets that would have been incongruent with the rest of the game style. (see Demo / Video)

Novelty and Uniqueness

- Procedural building of cities at runtime given a longitude and latitude coordinate chosen by the user. (see Demo / Abstract / Technical)
- Culling at runtime (see Optimisations)
- Having the capability to pick a real-world location to play in. (see Demo / Abstract)
- Created a tactile and kinetic movement engine (see Movement Engine and Demo)

Report and Documentation

- Created wikis for larger parts of the project. [2]
- Created architecture diagrams where relevant - pipeline, networking, and precompute [2]
- Created a comprehensive report that was 30 pages long, contained $>$18000 words and included graphs and diagrams.

Abstract

Overview

Elemental Explorers is a multiplayer virtual reality game set in procedurally generated environments based on real-world data. Two players compete against each other to race to points of interest (Figure 1), scoring points for being the first to arrive. Points are also scored by performing well in the mini-games that appear upon arrival at the points of interest.

Figure 1: A screenshot of our game being played in multiplayer

Story

1000 years ago the human race evacuated Earth due to the rapidly declining climate. The oxygen levels have returned to normal levels, but Earth is mostly uninhabitable. Buildings have been claimed by the hyper-evolved natural world in the time since humans left. It is your job to reclaim the lost buildings; race against your crewmate to recover the most. As a member of the Intergalactic Space Fleet, you have not been performing to the required standards. Your captain has decreed that whoever reclaims the least points of interest shall be left stranded on Earth!

Environment

The spaceship scene consists of a series of kit-bashed assets [3]. There is a holo-table with a VFX graph detailing a height map of the United Kingdom. The user is able to select a location on the map interface to play the game in. Selecting a location runs an asynchronous pipeline to procedurally generate the selected scene. Running in the background, this allows for seamless scene transition without a substantial drop in framerate.

The environment has terrain which is height mapped based on real-world elevation data. This terrain is overlaid with a water mask to map out the location of rivers, lakes, and other significant bodies of water. All buildings in this area are modelled from their footprints. Their building mass is procedurally generated (Figure 2) from the information obtained through various APIs. The road network has been reconstructed, although it is disconnected in places, as one would expect after 1000 years of erosion. There are natural assets like leaf clumps scattered on the building meshes. Grass and flowers grow on the terrain and through the roads.

Figure 2: A screenshot of our procedural generation of Bristol.

Gameplay Loop

The player starts off in the spaceship with their crewmate. The user can create a lobby and has the ability to select any location in the United Kingdom to play in. Once a location is selected (Figure 3), the players go into the elevators and collect their gauntlets in order to get their grappling powers. Whilst in the elevators the players are introduced to the story of the game and an introduction on how the gauntlets work. The players are then free to explore the ship's hull, containing a shooting range and obstacle course, so they can familiarise themselves with the movement system. This also serves as a time for the world to be procedurally generated in the background. Once the world has fully loaded the players can enter the dropship, moving them into the main body of the game.

Figure 3: A screenshot of our location selection holo table.

Once in the dropship, the doors open and they are given a countdown before they are able to begin their mission. Glowing chevrons direct the players in the direction of their first rendezvous location. These are locations that are particularly overgrown by nature and need to be rectified. Players can see their competitor through walls due to the thermal imaging capabilities of their HUD. The race is on to reclaim the landmark. When the players arrive at the location they must shoot the most hostile plant life, collecting the most

seed samples and ensuring they earn their spot on the Intergalactic Fleet. Once all of the buildings have been found, the most competent crewmate will be beamed up into the spaceship. The loser is left on Earth to fend for themselves and find their own way home.

Player interaction and Controls

Players can grapple through the environment by pointing at an area with either hand, pulling the trigger, and physically pulling themselves towards that point. The player receives haptic feedback if they have successfully grappled to an object with a collider, letting them know they can pull themselves in.

These grapples have a correction force applied to them so that if you are hurtling directly toward a building, your movement path will be altered so that it feels smoother. This reduces the skill floor, allowing our game to be more accessible, and helps reduce motion sickness. Similarly, there is a grapple
falloff curve that is calculated in a sliding window which prevents the user from spamming their grapples to maintain the maximum velocity. Your grapples' relative power is indicated to you by the colour of your gauntlets, which fade to red when you have minimum power.

We have also added functionality for joystick movement for fine-tuned movement. However, user testing prompted us to add teleportation as an alternative as it caused reduced motion sickness and allowed players to move vertically upwards (for example, to climb over parapet roofs).

Inverse kinematics is used to predict the player's arm positions so that player avatars mirror the player's movements and gestures. We decided to hide the player's body from the first-person perspective as there is not enough information to precisely track the positions which reduces the in-game immersion. The player's height is calculated using the position of the headset and is used to scale the player model accordingly.

To reduce motion sickness, moving in real life corresponds to moving in the game. This is natural to players and without it can be disorienting and sickening (as when leaning forwards you expect your point of view to change). This has the side effect of when a player tries to walk through a wall that exists in VR but not in real life, they are blocked from moving which is nauseating. This is a limitation with VR, but our implementation is the best solution we found to prevent players from clipping through objects whilst mitigating motion sickness.

Team Process and Project Planning

Coming into this project, we had a varying degree of Unity experience so we paired up on tasks in the beginning. Less experienced members contributed technical ideas from a more conventional computer science standpoint. Those more well-versed in Unity were able to translate these ideas into implementations whilst explaining the architecture and limitations of the Unity Engine. It is because of this technique that we were able to maintain a wealth of strong ideas and quickly acquire six team members capable of writing useful and well-written Unity code.

Weekly Meetings

The structure of our team was centred around having regular meetings and completing agile sprints weekly. Typically, we would book a meeting room on Monday morning and decide on the most important tasks that needed to be completed. We'd go over these in comprehensive detail so that everyone understood what was necessary for each task. Tasks were allocated based on individual interests and relevant prior knowledge.

We would assign follow-on tasks to people if they were able to complete everything quicker than expected. Usually, these ended up being the main focus for the next week's meeting, which meant that we'd had a reasonable amount of time to come up with ideas whilst working on other aspects of the game, giving us a great development loop.

In cases where the task was quite large and unable to be split into smaller parts, or these parts were tightly coupled, we split into small scrum teams of size two or three. We found that this was the most productive way of working on the project. This allows us to quickly solve complex problems that could take an individual a significant amount of time. Furthermore, multiple team members having knowledge of complicated parts of the game makes future debugging and integration quicker.

Another important aspect of our team process was the fact that all of our team were in the project room 4/5 days a week. This was crucial for our team to be an efficient production unit. Being in person helped to ask "silly questions" that could have otherwise taken hours to figure out. Similarly, we were able to utilize a practice known as "rubber ducking" [4] in which we would explain a problem we were struggling to solve in small logical steps. Hopefully, the care given in explaining this to another person who may simply ask the correct questions will bring the issue to light.

In addition, our team manager made sure that if somebody was spending longer than anticipated on a task, then we would assign someone else with a lighter workload to help with it and understand the issues they are facing. This could also happen following research conducted by a team member if they identified that a problem would be solved

faster by someone with a skill set better suited for that task.

Conflict Resolution

Disagreements are inevitable in large-scale group projects which led us to implement measures to prevent an unprofessional environment. Our team manager, Matthew Swann, established that everyone had a say in the weekly meetings so that no one felt that their opinion went unnoticed. When people's viewpoints clashed, they were each given a chance to explain their ideas. The group would then unanimously decide which idea to proceed with. In the case our group was split we asked our TA, Jordan Taylor's, opinion which was especially valuable as he provided an outside perspective. We agreed that contributions were to be voted on anonymously and averaged so everyone can be honest and maintain their discretion.

Agile Sprints

We divided each week of the term into an agile sprint. This consisted of splitting off into multiple scrum teams and working on a task until completion. We would assign people tasks every Monday, then hold a mid-week update on Wednesday with a revised expected completion date. If this deviated largely from the original date specified, such as coming across a problem nobody had seen before, then these problems were discussed. Finally, we would have an end-of-agile sprint meeting on Friday afternoons in which everyone updated the rest of the group on their progress for the week. In these reports, we explained challenges that we overcame so that people were cognisant of them. This would be helpful if they were working on that part of the system or a similar problem in another area entirely. It was also allowed us to identify when team members were not contributing effectively which triggered discussions on how this could be improved.

We allocated members a mixture of, tasks they have a particular interest in, and others that are important to progress the game. This reduced burnout because people were motivated to keep working on tasks they enjoyed whilst underpinning the project with a consistent structure. We were always working to add features that would lead to a complete, functioning gaming experience.

Planning

The planning for the project was split up into 3 distinct sections:

World Generation

Networking

Gameplay Management

Since we planned to procedurally generate the world, we decided to create a procedural pipeline that split the process into several stages (Figure 4). Thus, each stage in our pipeline could be thought of as a function that is applied to the output of the previous stage. Moreover, we planned

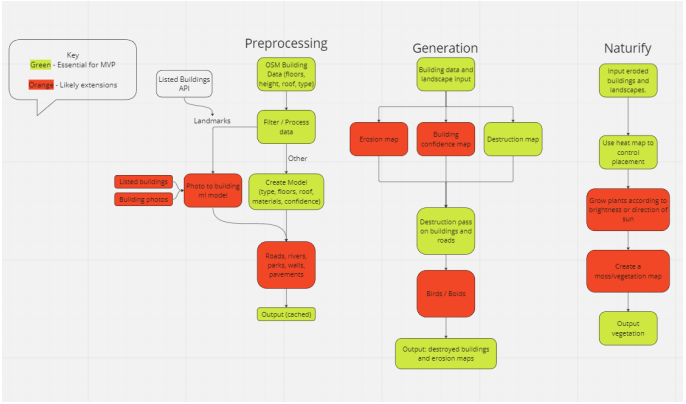

Figure 4: A screenshot of our original pipeline plan on a Miro board.

on having two main stages in our pipeline, reconstructing the world and then applying nature/ destruction to it. These later became known as the precompute and runtime pipelines respectively. We planned out our development for the pipeline, giving a minimal functioning product in time for the MVP. This could be built on, providing additional features for our beta and final release.

The original plan was for a race course to be procedurally generated in the city that was chosen. We designed our locomotion around the idea of a race. However, play testing revealed that if there was a reasonably large skill difference, the players only saw each other at the start of the race. This caused a disconnect between players, making it less competitive and enjoyable. As a result, we had a meeting focused on discussing a change in our core gameplay. We wrote fifteen ideas in length ranging from complicated asymmetrical cooperative games to popular party games [5]. In the end, we decided that multiple races to points of interest were more fun and take the players to regions that they were more familiar with. In order to make it more social, we planned on inhibiting a player that first reaches a point of interest to reduce the skill gap. This way there is always the opportunity to win so a bad start doesn't necessarily mean the entire game is unwinnable. We also planned on always being able to see players through the walls so that the players always had a reference to how close they were to each other.

Another example of how our iterative planning and development cycle benefited our final product can be seen in our building mass generation. Our first attempt at generating the building masses from data was to apply context-free grammars inside Unity, to place assets and apply textures/materials. While the results looked promising, we realised that insetting assets into the meshes would involve dynamically recalculating the vertices and triangles of the meshes. This would take 1-2 weeks of work and not increase the value of the game enough to be worth the time investment. Also, applying assets like chimneys, columns, balconies, and spires

would all have to be implemented in their own separate functions which was impractical given the time constraints of the project.
As a result, our lead designer did further research into previously discounted ideas. This resulted in the decision to run Blender in headless mode from inside Unity and use
Buildify [6] generators to perform the same task as the context-free grammars used previously. This also allowed us to use Quixel Megascans [7] which accelerated the creation of generators. Since each generator took about one day per person to make and integrate, this liberated more resources that could be used in other areas of the project. Overall, this resulted in a higher quality product that was cheaper to implement.
Both of these were examples of how our team utilized iterative development. When planning how to solve a problem, it is not always possible to understand the problem deeply enough to come up with a perfect or even viable solution. It is for this reason that we used lessons learned from previous iterations to improve the final outcome. As a team, we would not hesitate to deprecate previous code implementations. This meant that conversations about implementations not making the final release, and having somebody else improve a feature, were easy to have.

Integration

As well as the three deadlines (MVP, Beta, and final release) given by the unit director, we decided to have a fourth release; pre-release which we would release after the Easter break. Despite merging development branches regularly, releases demanded additional work and the incorporation of unfinished features.

For the MVP release, the team made a significant amount of progress on all aspects of the game. Consequently, the integration of features was a larger task than we had expected. As a result, we only finished polishing our release shortly before the panel was due to play it. In our post-MVP reflection, we decided that in the future we would have a finished build at least the night before any release deadline.
Integration before releases was also a chance for everyone on the team to learn about how the other parts of the project worked. For instance, our pipeline had numerous iterations which ranged from the original to a tiled version, one that loaded data from the disk to an asynchronous pipeline. Since the pipeline itself was a novel tool these points of integration were ideal for others to get a chance to learn how to use them effectively.

Between releases, we would write out a list of features in response to user feedback that we wanted to be completed before the next release. Some of these were optimistic or removed due to a change in design ideas but for the most part, we managed to implement all that we set out to achieve for these releases. Post-release, the development would generally diverge as people worked on distinct features. As we progressed, the development would become more coupled, culminating in a nicely integrated release with all features included (Figure 5).

Project Management

Initially, we used GitHub's Kanban board to assign and track tasks (Figure 6). This was useful since there was a large number of issues, and with the agile nature of our development process, we frequently created new ones. This allowed us to assign priorities and label tasks into similar categories. Similarly, we were able to provide detailed instructions on a task if we were assigning it to another person. It was easy to provide updates and have discussions on each issue.

As our project grew, the number of tasks listed become unmanageable as many subtasks were created and were listed alongside bug fixes and features. This made it hard to understand what was being worked on and which issues could be started on. To prevent these problems we switched tools to use Monday.com [8]. This was because it allowed us to display subtasks and model dependencies between tasks much more effectively. This made it easier to see the progression of tasks at a glance (Figure 7). Furthermore, it allowed us to set due dates so team members could easily see what everyone was working on and when they expected features to be completed, which was invaluable with our frequent integrations. We lost no functionality by switching, so for us, the clear benefits made the choice obvious.

Furthermore, we utilised GitHub Wikis to create documentation for our project. Once we had finished implementing a feature, it was required to complete a write-up including any architecture diagrams for that particular section of code. This meant that all team members were able to have a base level of understanding across all aspects of the project by keeping up to date with the pages on the Wiki. If someone didn't fully understand something, they could then ask further questions but this streamlined the process as the person did not need to be taught the entire concept from scratch. We had to make our GitHub repository private when we added Quixel Megascans assets. This meant that we lost access to GitHub wikis so we migrated to our own wiki site on Notion instead.

Before beginning a new task, we had whiteboarding sessions with two or three team members to make a full plan.

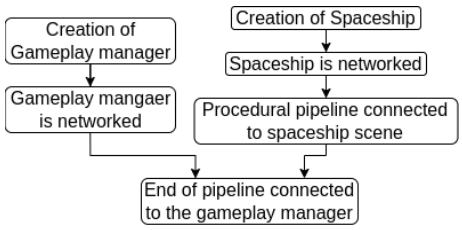

Figure 5: An example of our workflow. These are all components that were integrated for the beta release.

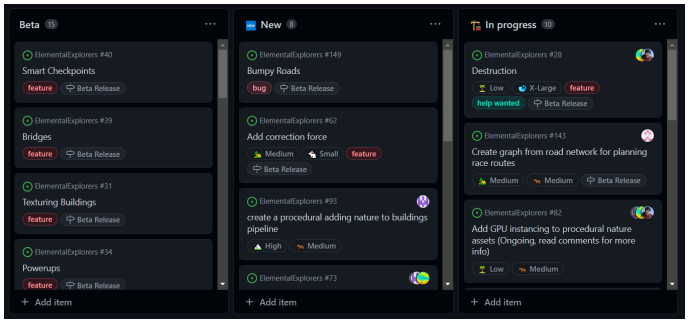

Figure 6: A screenshot of our Kanban board while in use.

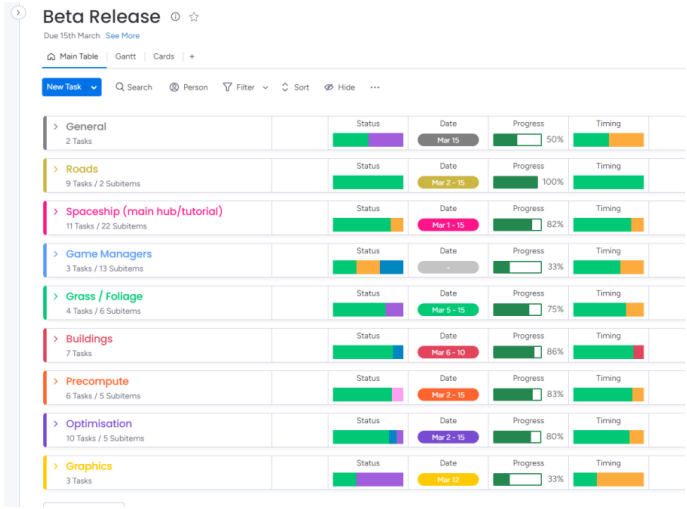

Figure 7: A screenshot of our Monday.com board while in use.

This helped us implement features more efficiently as they have been fully considered by multiple members of the team rather than implementing the first idea we thought of. We would discuss the issue at a high level and break it into smaller sub-tasks. This enabled us to put these in the correct format on Monday.com, giving the rest of the team an idea of the plan and how we are progressing. Similarly, completing subtasks is rewarding for individuals, increasing their motivation to continue working. This also prevented someone from continuing to tweak aspects that have a smaller impact on the game as a clear goal was defined for each task.

Throughout the project, we had multiple people switching to different aspects of the project. This prevents one person from working on the same task. This could become repetitive or result in them losing motivation. In addition, when a new person comes in and works on a problem for the first time they are likely to try and tackle it in a different way than others have so there are more ideas, and hence progress can be made quicker. For example, James had been working on roads for a week and when the task was reassigned to Matthew, he came up with a way to merge them together by converting it to a graph network. This solution also allowed us to perform various algorithms on our roads as discussed later in the technical section.

Reflection

Over the course of 12 weeks, we gradually improved our collaboration skills by learning from various complications and inefficiencies. After each release, we had a post-release reflection meeting and discussed what went well and what didn't go well and how we could address them going forwards. One issue we had was that some members worked on the same issue but had completely different sleep schedules. This meant that there was little overlap in the time that they could work together which would lead to inefficient uses of time. We solved this problem by decoupling the subtasks and making smarter pair assignments when distributing tasks. Decoupling subtasks allowed members to work asynchronously and they could update each other when their work schedules overlapped. We also only assigned scrums was a large overlap in working hours so that they were always able to communicate with each other.

We realised that it was important to have a clearly defined end goal when setting out upon completing a task. It was very easy to continue working on a feature past the point required for our game. We want features to be of a high standard but each feature doesn't need to be completed to a AAA-studio standard. The law of diminishing returns [9] combined with the limited time frame we have for this project meant that this was an important pitfall to avoid. At the start, we encountered this issue with our movement engine as it's always possible to think that you can improve the feel of the game and spend a large amount of time tweaking parameters. We made changes to our planning to prevent this from happening in the future, namely adding deadlines to tasks and breaking each larger task into a series of smaller sub-tasks. We found these to be effective enough

for our team and we didn't encounter this problem again.

Another problem we found was dealing with conflicting feedback from a diverse range of people. We received feedback about wanting to change the game from a race to something else, but others said that we should keep it as a race. We gathered everyone's assessments and concluded that we should prioritise drawing attention to our environment and building our game off that. We decided the best way to do this would be to make players race to notable points of interest in the area they selected. This incorporated aspects as both suggestions and improved the enjoyment people were able to get from the world we had generated.

We found regular meetings really helpful as they grounded all team members in the project and everyone knew what the rest were doing at all times. Moreover, everyone being in person on the days that we worked increased productivity greatly. Online meetings and communications via messages are just no substitute for in-person work. In addition, our iterative design process yielded a product of a higher standard that we wouldn't have come close to creating if we had a less adaptable strategy.

Individual Contributions - Matthew Swann

Initial Planning - 1 week

With the rest of the team, I spent the first week of the project doing research, learning unity basics, and planning our development.

Movement Iterations - 2 weeks

I worked on the first movement system using realistic rope physics which allowed players to swing with Imran. This required researching how to implement a VR controller in Unity and integrate it into a simple player structure. This took about one week. I also worked on the final movement iteration with George allowing players' hand acceleration to be used as input. This took a further 1 week.

Added reasonable player collisions - 1 week

Due to limitations with how the VR controls are represented in Steam VR, the player structure needed changing to allow the player to move in real life and have this translate the in-game movement. This prevents motion sickness and makes the game feel far more immersive and less motion sick inducing. This was particularly hard as we had to prevent the in-game player from moving through objects in the game when a player moves through one in real life.

Steam Input Wrapper - 1 day

I created a simple wrapper to convert the callback-based VR input controls into a simple library. This is because team members were used to the unity input system and creating a bridge between the two made our code much more readable and comprehensive.

Networking the player prefab - 1 week

I worked on creating the multiplayer and single-player prefabs so players could be controlled both when connected to a server and when offline. Due to restrictions in Netcode for Gameobjects, these need to be distinct but I implemented a proxy Player prefab in order to increase DRY code and game objects.

Creating the road network graph - 2 weeks

I worked on converting the old road lists into a network graph allowing more efficient algorithms to be run. This made the building of roads more consistent with fewer breaks between objects. I also created a restricted Dijksta's algorithm implementation which used the road network to direct the player in the correct direction.

Creating the race - 1 week

Before we removed the majority of the race functionality I implemented the gameplay loop. This took two locations in the pipeline and placed checkpoints between them going past notable points of interest. This was all networked and setup so that players times would be recorded with confetti fired from the final checkpoint when it was reached.

Converting the pipeline to be asynchronous - 2 weeks

I worked on re-writing the pipeline to allow asynchronous nodes and allow nodes to process work over multiple frames using coroutines. I also created a profiler to better identify slow nodes which lead to notable improvements in the pipeline runtime. I then integrated the pipeline into the game so that it was run when the players had selected a start location.

Terrain height sampling - 1 day

I implemented a function that more accurately sampled the height of the terrain. This led to multiple notable improvements in road-smoothing, road placement offsets, and better building placement. Before we were using ray casts which could not be run asynchronously and were not very performant.

Audio manager - 1 day

I created the basic framework for managing audio for the music and voice-overs in the game. This allowed other members of the team to trigger audio both locally and from all speaker objects in the scene.

Async Scene Loading - 2 weeks

I worked on connecting multiple scenes together asynchronously so that players transition without any notable changes. It also triggered any startup events in the next scene and persisted information across scenes and was fully networked without any desync.

Shooting Subgame - 1 week

I fully integrated and networked the basic sub-game developed by George into the main gameplay loop. This involved networking all the events using RPC calls and implementing a workaround to fix bugs in the unity version we are using for the project.

Updating teleport - 1 day

After user testing, we identified that users were struggling to use the teleport with its old implementation. Users expected the teleport location to follow the arc but this was not the case. I worked with Alex to implement a curved ray cast create that was affected by gravity for more intuitive controls.

Integrating others work - 2 weeks

As I have worked on both netcode and the main gameplay loop it made sense for me to help others integrate their own work. I have both independently and assisted in implementing other team members' work into the game. These features include:

- Vignette
- Particle effects
- Player rig
- Adding voice lines

Individual Contributions - George Wigley

Initial Idea and Procedural Pipeline Research - 1 week

After our project was confirmed, I spent some time researching potential technologies and implementation methods. This involved the idea of the procedural pipeline.

Initial Implementation of Procedural Pipeline - 2 days

Having pair programmed with James Millan and Stephen Brock, I refactored and continued development on a fully custom pipeline tool.

Researched xNode - 1 day

After the initial pipeline development efforts, I researched alternative systems that could be modified and adapted to our use case and came across xNode. I then tested it to ensure it could work.

Moss and Nature Assets On Buildings - 1 week

Created an editor tool in Unity that projects procedural Perlin noise onto it using its UVs. The system then samples points across the mesh in 3D and generates a set of spawning points based on the noise. This evenly generates points across any mesh contained by noise. The noise was also used to apply a moss texture.

Final Compilation of Photo2Building - 1 week

Having spent time attempting to compile Gen Nishida's Photo2Buildings program with James Millan, I was able to complete the compilation and evaluate the program.

Setup Quixel Megascan Materials - 3 days

Performed initial research into the usage of Quixel Megascans in Unity, this involved testing the bridge importer tool as well as constructing a Tri-Planar shader to evenly distribute the texture across meshes without distortion.

Player Controller V2 (non-VR) - 1 week

Created a non-VR player controller to test potential new movement methods, this involved a custom solution for friction as Unity's default solution was not appropriate.

Final Player Controller - 1 week

Worked with Matthew Swann to create a system for tracking the players' motions and use them for movement, this also encompassed a rewrite of the movement system.

Procedural Road Shader - 1 day

Created the first shader for procedural roads ensuring that dashed lines and textures would be applied evenly regardless of road shape or UVs

Modified Unity Provided Assets To Create Spaceship - 2 weeks

Using a Unity provided asset, I was able to modify both the meshes and VFX assets to create the spaceship used in the game.

Created Gauntlet Model - 1 day

Created the 3D model used for the player's gauntlet with an emission channel used the relay information to the player.

Created Tutorial Zone - 1 week

Created and textured a tutorial zone that allowed the player to test movement whilst the level loaded.

Created UI Material and Shader Pack - 1 day

Created a set of shaders and materials that could be dragged and dropped onto UI elements to enforce consistent styling across the game.

Research and Testing MCut For Destruction - 1 week

I researched a solution for performing boolean operations on meshes within Unity and determined after research that the C++ library MCut was our best option. This would require writing a C-sharp wrapper for the library that would accept Unity's mesh format. After initial research and experimentation, this was deemed too great a time commitment for the value it provided.

Added Collision Prevention Force and Anti-Spam - 3 days

Worked with James Millan to add a collision force to the movement system and an anti-spam system to reduce the skill difference between players.

Researched and Tested Blender Integration - 3 days

Researched the potential usage of Blender from within Unity. Also wrote and tested the integration between these tools before we moved to this new system.

Created Working Prototype of Blender Unity Integration - 1 week

Utilizing the blender tool Buildify, I experimented with initial building generators using geometry nodes. I also created a 32-minute tutorial explaining what I had learned to other team members allowing them to contribute to the system.

Created Building Generators - 2 week

Created generators for several building types.

Target Game - 1 week

Created the target shooting game including animations, textures, and VFX. I also coded the logic for generating target spawn positions.

VFX and Particles - 2 week

Created various VFX and particle systems throughout the game, the main example is modifying the unity code to create the holo table on the spaceship.

Interiors - 3 days

Followed a tutorial on how to create a shader to render building interiors from cube maps

Individual Contributions - James Millan

Lead Programmer - 1.5 weeks

As lead programmer on the project, I set up a linter and continuous integration on the project. In addition, I created a git contribution guide that improved the efficiency of our development by reducing the number of issues we had using collaboration software. Similarly, whenever anyone had an issue with Git I was able to give them the relevant commands or workflow to fix it.

When researching how to use Git with a Unity project it was recommended to use Git Large File Storage (LFS) and a .gitattributes file was given. However, we discovered that this file added many files to LFS since it is pattern matched against nearly every file type. This meant that very quickly, we went over the default amount of bandwidth provided to us by GitHub. Removing LFS files is quite difficult so every file had to be untracked, and removed from all caches in the .git folder then recommitted. A better solution would have been to manually add the file paths of files with size $>$ 100MB to the .gitattributes file.

Procedural Pipeline - 1 week

I was involved in the creation of the first pipeline. This involved extensive debugging and the creation of several nodes, iterations of which were used in the final pipeline. Similarly, I maintained nodes so that when features were extended, the nodes still received the required inputs and produced the correct outputs.

Building Reconstruction - 2 weeks

Worked with Stephen Brock to reconstruct the buildings from the Overpass API. This involved understanding how the API worked including the format that OSM data is stored in. Also, finding a provider that had a license that met our requirements. Then implementing how to acquire all of the ways and the nodes of these ways. The same for relations that have inner and outer ways. Also solving the building:part situation with the offset calculation. There was extensive debugging completed in this area by trying the building reconstruction on multiple different bounding boxes across several countries.

Roads Reconstruction - 2 weeks

Responsible for the initial version of roads that were disconnected ways. This involved querying the Overpass API and retrieving the extra nodes in batches by their IDs. Also, I researched asset libraries that created procedural road structures and altered Sebastian Lague's Path Creator [10] to procedurally generate the road meshes from Bézier paths.

Building Mass Generation - 2.5 weeks

Implemented the application of context-free-grammars for building mass generation. This involved creating grammars and creating a recursive descent parser for each one. Also, assigning a grammar to each building and applying windows and doors to them in a procedural manner. Moreover, I created procedural triangular prism roofs. I created procedural probability distributions that updated the probabilities of applying various assets based on building type.

Gameplay - 2 weeks

Implemented various movement improvements: correction force, grapple falloff curve to prevent spamming and combo system. Worked on the new gameplay manager and gameplay loop. This involved the creation of a points of interest (POI) node that was first called an API, if no results were returned, calculated the areas of buildings and returned the largest building. So a POI was always guaranteed to be spawned. This involved networking several parts of the gameplay manager. In addition, I extensively play-tested the movement

Research and Planning - 1.5 weeks

I researched the compilation and use of Gen Nishida's Photo2Building for use in this project to generate complex building facades for important buildings. Also, I researched the use of context-free-grammars and subdivision for building mass generation in our project. In addition, I researched the Overpass API, looking into how to retrieve roads and buildings as well as what tags are relevant for assigning generators. I planned out the creation of the pipeline, building, and road construction and the new gameplay manager.

Report and Documentation - 1.5 weeks

I took the lead on writing the report, writing the team process, software and development, nine aspects, the abstract, and some parts of the technical sections. This took approximately 1 week of work. Furthermore, I wrote documentation for all parts of the project I worked on.

Individual Contributions - Stephen Brock

Initial Procedural Pipeline Implementation - 1 week

I implemented the initial manager of the synchronous pipeline, which controlled the logic for creating inputs, outputs, and running the pipeline. This also included extending the base classes for the nodes to be inherited.
I later helped to convert some of the nodes to be asynchronous when the pipeline was changed.

Terrain - 1 week

I used data from APIs (see Terrain Generation) to elevate the terrain according to real data and mask the water.

Multi-tile - 2 days

I pair-programmed the multi-tile implementation with Imran. Wrote post-pipeline implementations to stitch together multiple terrains and global data.

Initial Building Mass Generation - 1 week

Helped research OSM data and the structures required to fetch data from the API. Created the initial 'blocky' meshes by triangulating the footprints and walls, including procedurally UV unwrapping them.

Grass - 3 weeks

Experimented and wrote the architecture for instancing grass. Created a compute shader for placing grass with a density map, reduced density with distance and accounted for it with scaling/rotating, wrote occlusion/frustum culling, and sampled in a way to reduce overdraw. Added a system for adding any amount of LODs for grass that varies with distance. Created a shader for grass with texture atlases, translucency, injected instancing, vertex offsets for wind, and various methods to reduce aliasing (see Grass and Asset Scattering). Includes counting compute shader for instancing in VR.
Created compute shader for generating a wind texture, applying it globally for any shader with wind implemented.

Asset Scattering - 3 weeks

Created general architecture for instancing any mesh with predefined positions, including the general compute shader for culling them with general instancing shaders. Chunked so only nearby instances are considered, and added support for two LODs. This includes implementations in the pipeline to create instances and output them.
Density maps were also produced in the pipeline, and I made several nodes and compute shaders to generate these. A node was created to render a mask of any particular layer (e.g. buildings). A compute shader and node were created for general noise textures with multiple octaves, one for calculating the distance transform of a mask to change density with distance, and a variety of nodes for simple operations to textures (e.g. subtraction, multiplication, threshold). Models and parameters changed

Optimisation - 2 weeks

Helped research and implement InstantOC with George, and set up an asset structure to work with the pipeline. Created various nodes for optimisation: a node to merge all meshes by chunk, a node to generate texture atlases per chunk, and a node to apply the InstantOC component to the GameObjects.
Created a precompute node to filter building parts that are under the terrain or within another building to reduce geometry.

Buildify - 1 week

Wrote the Unity-side implementation for Blender's Buildify. This included serialising JSON footprint data for input, reading the JSON output, and mapping it to the correct prefab. I made two building generators, one for a large retail building and another for an office. I also worked on integrating created buildings into the correct structure to work in Unity.

Fog and Clouds - 2 days

I converted shaders I created from previous personal projects for the clouds and the screen-space fog to work in URP and VR.

Individual Contributions - Imran Zamin Ali

Planning - 1 week

The first week consisted of almost entirely team meetings, research, and planning.

Initial VR setup and controls - 1 week

I initially worked on setting up the Valve indexes. This required me to research the Steam VR plugin and set up the controller mappings. I also implemented haptic feedback and initial joystick controls.

Movement - 3 weeks

I developed iterations of movement with Matthew Swann, using the Unity Physics Engine, allowing the player to both swing and grapple. After an iteration, I would invite people to play-test and attempt to incorporate their feedback. I also lowered the skill floor by implementing lock-on, aim assist, and limiting the max speed.

Pavement and road shaders - 4 days

I extended George Wigley's road shader and combines the two to represent roads with pavements. I then made them appear damaged by imitating cracks and potholes.

Water shader - 2 days

I replaced Stephen Brock's water shader with one that faked movement.

Water penalty system - 3 days

I added a penalty system to incentivise the player to follow the roads by slowing them down on the water.

Precomputing tiles - 1 week

I co-wrote a system, with Stephen Brock, that precomputed the building and terrain data which we then saved and loaded from a binary file. We then implemented a method to stitch the terrain together after the pipeline has finished running.

Flask server - 1 week

I implemented a Flask server to store the precomputed tiles. I added functionality to get a list of available tiles and another to request the tile data itself.

Location Selection - 2 weeks

I made two iterations of location selection. I added functionality to zoom in and out and provided a shader to highlight precomputed areas. I then added functionality for tile selection, taking the four closest tiles to the start position.

Vignette - 2 days

I researched ways to reduce motion sickness and decided upon adding a vignette to restrict the player's peripheral vision. I then created a controller for it allowing the user to have it off or on three different strengths.

Sound effects - 1 week

I made my sound effects using foley material we recorded ourselves in Audacity. Then I applied various effects to try to produce the desired sound for our game. Unfortunately, not all of them were successful and did not make it into the game and I ended up using royalty-free source material instead.

Benchmarking and profiling - 2 days

I used Matthew Swann's code to benchmark the MVP, beta, and final release pipelines so that we can make comparisons in load times. I did the same using the Unity profiler to analyse optimisations and uploaded my findings into a spreadsheet to analyse the findings.

Inverse Kinematics (IK) - 3 days

I researched using IK for our character model to create natural and dynamic movements. Using Unity's Animation Rigging package, and a custom controller script, I attempted to implement this. However, due to the absence of full body tracking, I was unsuccessful in making a stable model using IK, and therefore George Wigley, who had more experience in this area, took over and made an alternative.

Individual Contributions - Alex Elwood

Planning - 1 Week

The first week consisted of almost entirely team meetings, research and planning.

Initial Network Solution - 1.5 Weeks

I began the project by researching the differences between Photon Unity Networking (PUN) and Netcode for GameObjects (NGO) and concluded that we will use NGO to support the game's multiplayer features. After this, I conducted more research to determine the network architecture and resources we would need to support it. I concluded that we would want a peer-to-peer network with a relay server to route communication.

Firstly, I created a Unity Gaming Services (UGS) organisation to allow all team members to access our cloud resources. Then I enabled NGO and Unity Relay for our game ready for networking development.

Created Lobby UI and backend - 2 Weeks

I created a UI for a lobby system designed to connect two players, enter them into the same scene, and start the game. As a new user of Unity, there was a steep learning curve but it provided a good opportunity to learn a wide variety of Unity features and become a proficient user. Once the design of the interface was created and a basic version was created, I pair programmed with Matthew Swann to create a system based on callbacks that supports UI interaction by using the controllers as laser pointers.

After finishing an interactable UI, I implemented the functionality by writing code to interact with NGO and Relay. I then pair programmed with Matthew Swann to enable the player prefabs to support network interaction. Due to the significant learning curve, this was a large undertaking but has allowed me to gain a high proficiency in this area.

Networking Rework - 2 Weeks

I significantly improved the robustness of our networking system by embedding a framework that uses a polymorphic Finite State Machine (FSM) to uniquely handle networking events based on the current connection state.

Integrating Lobby System with Spaceship Scene - 2 Weeks

After George Wigley created the spaceship scene, I integrated the existing lobby system with it. This allows players to connect to each other before being able to transition to the tutorial zone via elevators. I then enabled the functionality of the dropship which allows the player to transition to the game scene.

To improve the feel of the game I added the 'spaceship AI' voice lines to the scene which are triggered at various points throughout the scenes.

Seamless Scene Transitions - 2 Weeks

Modified the existing Scene Manager to handle seamless scene transitions. These involve having new scenes loaded in the background and re-positioned so the player moves between them without an indication of entering a new scene. Once the player has moved away from the old scene, it is unloaded in the background.

Once the procedural pipeline was made asynchronous, this had the added complexity of having to manage the large duration of the scene loading. Much of this was pair programmed with Matthew Swann.

Procedural Building Destruction - 1.5 Weeks

As it was a major component that I had not yet interacted with, I began by gaining an understanding of how the procedural pipeline worked. Once I was confident in my understanding, I created a node to take generated buildings and apply a Constructive Solid Geometry (CSG) algorithm to them with the aim of producing convincing destruction. However, after extensively trying a variety of CSG libraries (Realtime-CSG[11], Parabox-CSG[12] and Unity Probuilder[13]) it was determined that none were viable options as they could not meet our robustness requirements. As creating our own CSG algorithm or attempting to improve the existing libraries would be too time-consuming we decided not to continue with procedural building destruction.

User Experience Improvements - 2 Weeks

I created a user feedback form which was used multiple times in user testing sessions. Particularly it helped in understanding the changes that needed to be made after major updates to the movement system.

One piece of user feedback that we received was that our joystick movement system was causing motion sickness. Taking inspiration from other video game movement systems[14] I added a point-and-click teleport movement system which made a tangible impact on the reduction of motion sickness experience. After some additional user testing, I pair programmed with Matthew Swann rewrite the system to improve its feel and increase robustness.

Another piece of feedback that we gained was that we needed to improve the quality of interaction between the two players. I helped to address this issue by adding a voice chat to the game by integrating Unity Vivox[15].

Software, Tools and Development

Development tools

Along with project initiation, we decided that it was important to use development tools to aid integration and code quality. We used Super-Linter [16], GitHub's built-in linting workflow, so that we could enforce consistent coding standards across the project. This ensured that all team members followed the same coding conventions and style, which made it easier to read and maintain our code base. In addition, this linter provides automated collaboration, running further checks on all pull requests. The linter can detect certain errors or issues in code such as unreachable code or the of dereferencing null pointers. This makes it easier to catch small problems before they become larger issues and increases the overall quality of the code base.

For similar reasons, we used continuous integration for our project. CI gave us the ability to automatically build the Unity project and run all tests in edit and playmode before merging in a pull request. This allowed us to find and fix issues in the software early in the development process. Similarly, when it came to integration close to a deadline, a pull request that resulted in the project not building could have been catastrophic and CI prevented a merge that contained such errors.

Development Process

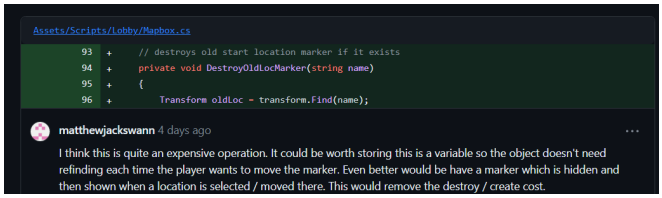

Figure 8: A screenshot of a requested change on a pull request.

Our development process followed an agile git-flow style. We would create feature branches off our dev branch each week and work on those features until completion. Once a feature was complete, we would make a pull request (Figure 9) to the dev branch. To make a pull request, a team member would create a list of all the changes they had made and link the issues this pull request would close on the Kanban board. The review process would consist of at least two other team members reviewing the code and testing the listed changes locally in the Unity editor. They'd also ensure that the workflows were executed successfully (Figure 10). If there was anything that needed changing, then the reviewer would request changes to be made (Figure 8), which blocks the pull request from being merged until they are implemented. For larger features like the spaceship scene, we branched off of the spaceship branch and then merged everything back into the branch before merging into the development branch.

Software Produced

A fundamental goal of the project was to be able to procedurally generate towns and cities from real-world data. For development to be effective and for it to be possible to perform these computations at runtime, we created a pipeline tool. The system allows for sequential computation, with results forwarded to the next stage in the pipeline. Results are also visualised in the Unity editor, providing a useful debugging tool. It gave us seamless integration for independent parts of the world generation and provides a useful way of testing how a change affects the rest of the environment.

To produce photo-realistic geometry that would improve the look and feel of the environment, we had to perform operations on the meshes we were creating in Unity. However, this would require re-triangulating and adding new vertices to the meshes manually each time we altered a mesh. This was a problem that had been solved already by Blender so we decided to experiment with running Blender in headless mode from inside Unity. We found that using the Blender Python API [17] we could programmatically perform these operations on our meshes and transfer the results into Unity. Since we are using Buildify generators to generically model building types, we produced a framework where, given the relevant Blender knowledge, anyone on the team could create and implement a generator for any building type requested. This was supplemented by a video tutorial created by our lead designer [18].

A large number of our world's assets are taken from online sources and not produced by ourselves due to the time constraints of the project. Most of this was the foliage and other scattered assets, so a general system to efficiently instance these dense repeated meshes is needed. A variety of methods were tested, with indirect instancing being the most frame and memory efficient by keeping all data and processing on the GPU (See Grass and Asset Scattering). With two custom compute shaders for culling, five buffers to properly copy over the data, and a custom shader to set the transform of each instance, a layer of abstraction was needed for this system to be used as a tool for any designer. The pipeline was therefore extended to allow instance outputs, which given a reference to an instancer and a list of transforms, would cull and render these instances in real-time. The instancer itself is easy to set up, needing only a mesh, a couple of culling parameters, and a material. As the material needed to be made specifically for indirect instancing, a variety of shaders were created. As a result, this lowered the barrier of entry for this way of efficiently instancing assets and allowed many team members to implement asset instancing in different areas across the project. Similarly, because of the documentation provided with this tool, a person that hadn't been part of this team previously would be able to utilize it.

Further Contribution

Definition A **well-defined input** is a data type either built-in to Unity or defined by the developer that is marked with the [System.Serializable] attribute.

To streamline the process of contributing to the code we ensured our codebase is well documented (Figure **??** on our

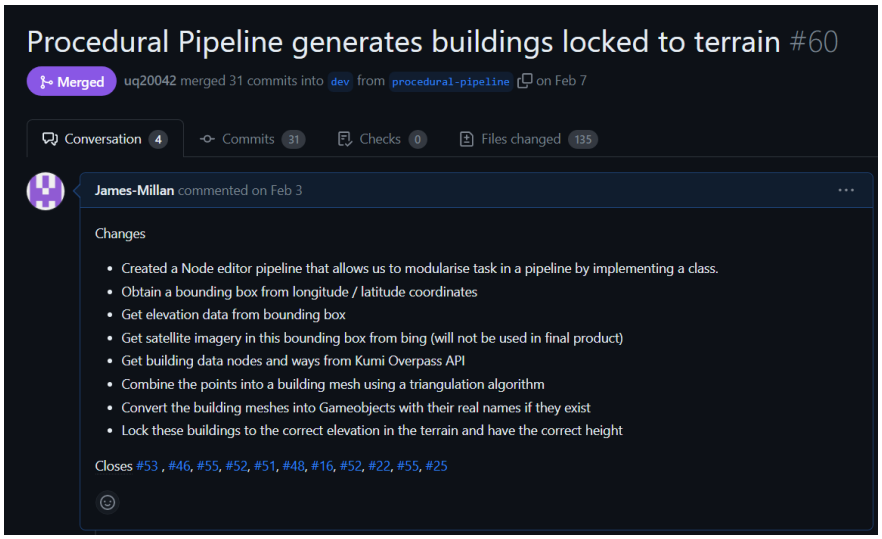

Figure 9: A screenshot of one of our pull requests created on a feature branch.

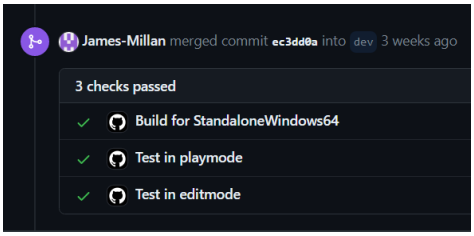

Figure 10: A screenshot of one of our pull requests showing the build and test workflows passing.

wiki pages. This would help newcomers to quickly familiarise themselves with our architecture such as extending or creating a new pipeline.

Our pipeline is a core abstraction, hiding the complexity of passing data throughout our codebase. By representing the pipeline as a DAG it is easier to see how smaller subroutines work as part of the larger system. Due to the modularity of the pipeline, contributors can easily extend one by creating new nodes and providing well-defined inputs and outputs with the exception of leaves. Nodes are flexible and allow for custom datatypes and can execute any logic required.

The game manager was designed to have no coupling with the world generation system allowing it to be replaced with a new one without needing to make compatibility changes elsewhere. This means changing gameplay can be done independently and will have no unwanted side effects on world generation.

Software Maintenance

The entire environment was procedurally generated given one pair of geographic coordinates. This meant that the best way to test our procedural pipeline was to run it in several different areas with some degree of overlap in the type of data we would get. Similarly, we would ensure that we wouldn't test using exactly the same coordinates each time so that we would avoid solutions that cater to these specific environments. An example of where this rigour in testing our pipeline came to fruition was with our procedural triangular roofs. When we ran it on places in the UK, we found that the initial solution worked well on these buildings. However, we found that in Barcelona the triangulation code wasn't working correctly and there was some flickering which was quickly resolved.

We considered writing unit tests that check each building footprint for a roof game object. However, this would have taken considerable amounts of time and was therefore deemed too impractical. The number of triangles and vertices were identical in all cases so unit tests would have had to go beyond checking values of properties of the game objects but had to check visibility at various positions in the scene. Given the iterative, agile nature of our development this would have slowed down our process and limited the quality of the environment we were able to produce considerably.

Due to complexity and high coupling, as the project grew maintenance of the networking code became increasingly challenging and time-consuming. It would seem that every bug fix would cause a different bug in a separate section

of the program. To overcome this challenge we entirely rewrote networking by integrating a Finite State Machine (FSM) [19] into the program where each state is a polymorphic network handler designed for a stage in the network life-cycle (i.e. HostingState or ClientReconnectingState). The advantage of this solution is that it leaves no logical holes in the handling of the entire game life-cycle and decreases coupling. Following this change, we found that subsequent updates to various aspects of the game caused much fewer network-related bugs, thus making maintenance of the game much easier.

As we kept adding nodes to our procedural pipeline it started to take prohibitively long to execute. It was at this point that we introduced the precompute pipeline and one that loads in the environment from a file saved on the disk. This meant that we could run the longer precompute pipeline at the start of development. When we wanted to test a new development, we would run the load pipeline, which was much faster, and saved us a lot of time debugging.
When we wanted to load in the world at runtime, we found that instantiating thousands of game objects as well as all the other expensive operations made the frame rate plummet to zero. As a result, we chose to make the pipeline asynchronous and implemented batched instantiation. This ensured that the frame rate wouldn't drop for a user in VR and break the immersive experience. This highlights how we maintained our tools in accordance with the needs of our development team and the needs of the game itself.

Technical Content

Movement Engine

From the inception of the project, we were aware of the importance of a strong movement solution. Our first steps towards creating a movement system were to review other games and solutions available and perform informal analysis to identify strengths and weaknesses. In order to get a better understanding of what type of movement we wanted, we tested multiple VR games with similar movement styles. Through this, we discovered how challenging a problem this is as many games has movement which has a high skill ceiling or is not very enjoyable.

Having an initial idea of what we wanted the final movement to feel like, we designated part of the team to be responsible for developing this movement system. The first concept was to use a combination of a physically accurate pendulum motion and a reel-in motion to provide the feeling of swinging across a city. This system involved the creation of a pendulum joint between the player and the end of a ray cast fired from the player's hand when they pulled the trigger. Whilst functional, this system had several drawbacks, primarily it did not provide the necessary control, as perfectly executing a swing following pendulum motion provides too steep of a learning curve (especially whilst moving at high speeds). In addition to this, the pendulum motion made many playtesters feel motion sickness.

Having reflected on this initial prototype, we experimented with variations of physics parameters. One of the more promising solutions was to use constant acceleration formulae to change the players' landing position to the target point. This created a parabolic path from the player's location to the grapple location (and was inspired by the doom meat hook [20]). This solution, whilst improving the control of the system, made the movement considerably less enjoyable and more motion sickening. At this point, we held several team meetings to discuss the overall goals and outcomes of the movement system going forwards and distilled these discussions to a few key points:

- The player's intentions should mirror what happens in-game as much as possible.
- We need to make more use of the VR aspect of movement, the movement system should be bespoke to VR and not be similar to other solutions in non-VR titles.
- There should be appropriate mechanisms in place to aid the user experience, for example preventing collisions and loss of rhythm.

Going forwards with these points in mind we changed the movement system to be more kinematic. The movement should actually use the player's physical motions to drive motion. Previously, we used a point-and-click approach to the player movement where very little hand motion was required to move. This came about from traditional video games which don't have the capacity to utilise actual player motion fully. We decided instead of using the VR controller as a means of aiming we should also use the players hand movements as a driver for in-game controls.

We tracked the player's hand movements to create a buffer of previously predicted velocities. We used this to derive an acceleration (See video [21] - 0:47) which was used in movement scripts. Acceleration with a magnitude over a certain threshold would apply a force in an opposing direction. This gave the illusion that the player was physically pulling themselves through the environment. This was then combined with the targeting raycasts from the previous movement systems to give the players a direction to pull towards.

We found this provided unparalleled control compared to our previous solutions and also greatly reduced motion sickness as generally, the player was moving in a straight line. After this system was implemented, we reflected as a team and agreed that it had met all three of the above-mentioned outcomes and so decided to confirm it as the final movement system.

At this point, we did more play testing and informally evaluated our movement system to discover any drawbacks and weaknesses. The major feedback that was unanimous from playtesters was that crashing into walls due to a poorly timed or aimed grapple was disappointing and potentially sickening. To remedy this we looked to other games to identify their solutions for players making mistakes that affected the flow of gameplay. After much research, we identified that

many games [22] used corrective forces to assist players in difficult tasks. Our answer to this was a corrective force to redirect the player away from walls if a collision is inbound. This was accomplished by firing a ray from the player in the direction of their velocity and computing any intersections with the level. Depending on the distance of the intersection from the player, we would provide a force to the player in the direction of the normal of the intersected face. This had the effect of cushioning players from collisions and also allowing players to grapple with a single hand without hitting walls. We used a curve that would map the distance before the collision to the corrective force applied, allowing us to define a function making the force feel natural to players. This was an essential addition as we wanted players to feel powerful and so needed to minimise their mistakes, however, doing this subtly is necessary for players to feel fully in control. Furthermore, players realised that they could maintain maximum velocity by repeatedly grappling rapidly. This lead to a repetitive game mechanic that wasn't very interesting for players. To disincentivise the player from doing this, we made the grapple's strength decrease if used too quickly. This was implemented using an animation curve that counted the number of grapples in a sliding window and applied a falloff multiplier. An animation curve is a tool that allows us to model complex curves visually without having to derive the function or its parameters by hand hence this expedited the development process [23].

User Testing

User testing has been a key element in our iterative development process. Our methodology consisted of conducting user testing sessions after every release in order to inform the next release iteration. Additionally, we conducted ad-hoc sessions whenever making a significant update. Almost all of these sessions were moderated, with a member of the team available to answer questions and to guide the participant as they play. At each stage, we had regular and new testers. This allowed us to both evaluate improvements made to the game and understand the current implementation from a fresh perspective.

User testing has informed the direction of much of our game, but in particular, it has revealed two major issues with our game.
Firstly, the first version of our movement system, although highly physically realistic, was not well received by new users. Players of all skill levels found movement unintuitive and struggled to maintain height and momentum. This led to a complete rewrite which was well-received by users (Figure 11).
Secondly, our initial racing gameplay was not well received by users, mainly due to the fact that players with high skill differences became too separated. However, highly skilled players really enjoyed the racing system. Therefore, we came to a compromise by pivoting the gameplay to a two-part race style where you have to race between each subgame and allow the slower player to catch up.

Toward the end of the project, we ran some unmoderated

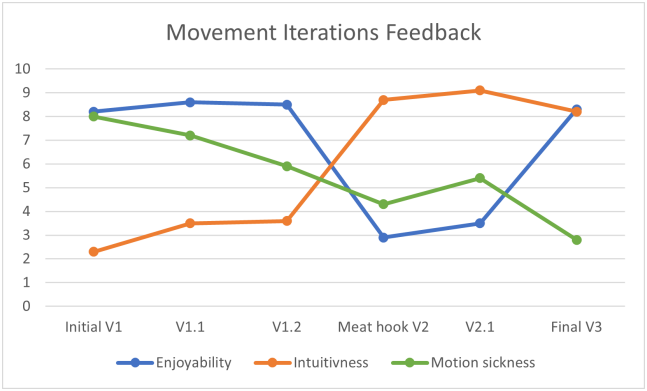

Figure 11: User feedback on the movement system over the scope of the project

testing sessions where the participant would not be able to ask questions or be guided by a team member. We did this because we wanted to test if the level of in-game guidance would be sufficient to act as stand-alone instruction for the game. This revealed that overall the gameplay loop was intuitive and well explained, however, some specific actions were ambiguous. For example, users did not have enough in-game guidance on which elevator to go to in the spaceship scene. This was fixed by adding additional voice lines, instructions to each screen, and impassable force-field barriers to the wrong elevator.

Procedural Pipeline

Early in development, we recognised that the amount of processing required for loading the world would lead to a huge amount of dependencies and a complex code structure. We decided to use a pipeline (Figure 12) to model the flow of data in a structured node graph. This allows team members to use nodes to perform actions on data, making it more readable with each node defining input and outputs without needing to understand the implementation used.

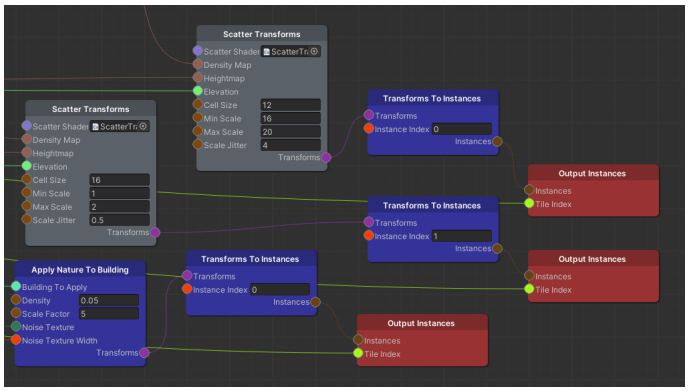

Figure 12: A section of the procedural pipeline.

The pipeline is a directed acyclic graph (DAG) comprising of input, output, and standard nodes. Computation is done

by calculating the outputs on all nodes with an incident degree of 0. These are then removed from the graph until all nodes have been executed. Input nodes take values specific to each run such as start location and output nodes are used to save the output for other scripts to reference. This allows the re-use of node outputs and ensures that all dependencies of a node are resolved before it's calculated.

This tool is a Unity editor extension that allows the construction of DAGs to visualize, debug and implement procedural pipelines. It is built using xNode [24] and all the inputs and outputs have to be Serializable. This means that a contributor can create their own custom classes and structs to use custom input and outputs. The first iteration of the pipeline constructed the roads and building meshes, created a correctly height-mapped terrain, and then instantiated these GameObjects onto the terrain. This included performing all of the slow API calls, some of which took upwards of ten seconds. Similarly, each node stored all of its inputs in an internal state. This meant that as the pipeline grew there was a greatly inefficient memory overhead that began to lag the Unity Editor.
We split the pipeline into 3 separate pipelines which are run at different times. As the amount of computation required grew, running the pipeline during the game became unfeasible so we introduced the precompute pipeline. This makes the API calls, reconstructs buildings, processes elevation for each tile, and saves the results in a custom file. This file can then be read later to retrieve the data in a fraction of the time. These files were hosted on a Flask server [25] so that players can request only the required locations at runtime and massively speed up loading. This also increased the reliability of the pipeline as we noticed the external APIs were not consistently available. The main pipeline reads these files and builds the world from them.

We decided to split the world into tiles so that we could easily track which regions were precomputed and represent them with a set of tile coordinates. The world is constructed from four tiles and the procedural pipeline is run once per tile. Their results are then stitched together once all their computation has finished. This brings the motivation for the final post-pipeline pipeline. It runs after all the tile pipelines have run, and contains logic that can only be run in a fully assembled world. This includes the large tree placement and road reconstruction (so that roads connect across the tiles).

The pipeline required optimising so that it can be run while the users are playing without incurring a significant drop in frame rate. First, we implemented a system that clears the pipeline, massively reducing the memory overhead. Then we changed nodes to calculate their output across multiple frames using coroutines. This allows nodes to monitor the current frame rate and do more or less work depending on how close to the target frame rate the game was running. We next implemented asynchronous nodes, which allow computation to be done on a new thread so that nodes run in parallel with one another. Due to limitations in Unity where specific functions can only be run on the main thread, only some nodes could be converted. This led to a huge performance increase for nodes which could be made asynchronous, however. We then updated the pipeline runner so that the use of asynchronous nodes was maximised. Our initial pipeline runner ran the pipeline in layers such that each layer is only run when the previous layer had fully completed. This could lead to conditions when asynchronous nodes had all their dependencies resolved but they still had to wait for the previous layer to finish completely. We changed it so that any asynchronous nodes that could be run would be started, and if none were available the synchronous node with the greatest depth would be started. This improved the performance of the overall pipeline further as can be seen in Table 1.

Pipeline Version	Runtime(s)	Minimum Framerate(fps)
Fully Synchronous	40.88	0.256
Multi-frame Nodes	43.93	57.37
Asynchronous Nodes	23.15	58.93
Improved Run Order	19.52	58.27

Table 1: Comparison of runtimes and framerates on a test pipeline with different pipeline runner versions.

Building Reconstruction

The structure used to represent buildings on Open Street Maps (OSM) is called a 'way'. These are a sequence of nodes which may have tags that contain additional information. A node is simply a longitude-latitude coordinate. A query to obtain this is simple and just requests all ways and nodes in a bounding box with the building tag set. Since each way has the IDs of all the nodes that it requires, any missing nodes can be obtained by requesting them by ID from the server. After converting longitude-latitude coordinates into Unity coordinates, a building footprint can be generated from a ways nodes.

In order to model the concept of complex building structures, OSM uses relations which are a sequence of inner and outer ways (Figure 14). Outer ways denote the general area of the building. Inner ways denote the area of the building that should not exist and should be cut from the original mesh created by the outer ways.

Midway through the project, the dataset we were using for Bristol got updated to use " Building:part " ways which were children of a parent way. Therefore, we had to update our structure in a way to include building parts and send off an additional query to obtain all building parts in the bounding box. A y-offset for each part had to be calculated so that it didn't clip with the way underneath it, in the case the building part was placed on top of an existing way (e.g.

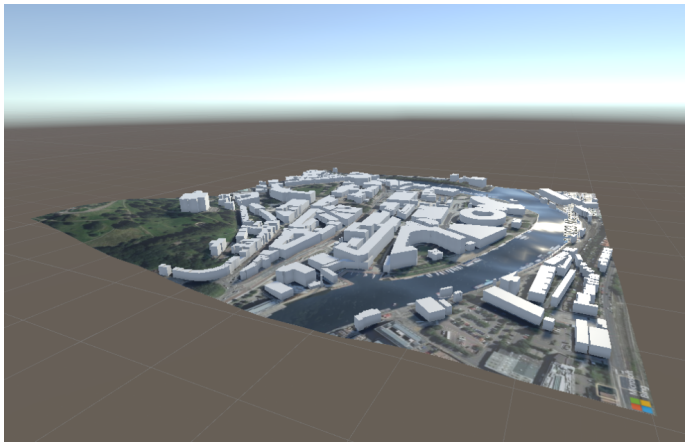

Figure 13: The initial reconstruction of buildings.

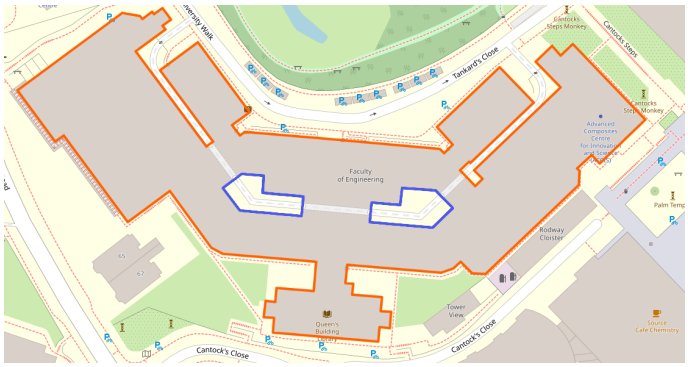

Figure 14: An example of an OSM relation. Outer ways are shown in orange and inner ways are shown in blue.

Wills Memorial Tower).

Ways and relations most commonly have tags that can describe many details about the building. These are used to assign building types in the cases that they exist, the most commonly used ones are the building and amenity tags.

Given outer ways and inner ways, we converted these into 3D meshes representing a general shape of the building (Figure 13). There is no standard order in which the ways are given, so first the way is checked and reversed if not anti-clockwise. To create the walls, the way is then looped through, creating two vertices and a triangle for each node. The same applies to inner ways, except the winding order is reversed to face inward. For the flat roofs, we initially attempted our own implementation of ear-clipping to triangulate any given polygon [26], however, we later opted for Sebastian Lague's implementation for its error checking and general robustness [27].

Each node in the way has an associated elevation, however, these elevations had to be flattened to the minimum value for each building. This avoids sections of buildings floating on concave areas of terrain.

Figure 15: Roads before and after merging.

Road Reconstruction

Similarly to buildings, the structure used to represent roads on Open Street Maps (OSM) is called a way which is a sequence of nodes. A node is simply a longitude-latitude coordinate that may or may not have tags that provide information about the type of road. There exists an additional structure called a relation which is a sequence of ways used to represent particularly complex roads. However, because of the format of data provided, the multiple ways in which these relations would have had to be merged together. Therefore, there wasn't any reason to request relations from the server and instead, implement a custom merging algorithm.

Roads within a given bounding box can be retrieved from an Overpass server using a simple query that requests all ways that have a highway tag set. Unfortunately, the only road nodes that have highway tags are things like traffic lights and bus stops etc. This means that a simple query for acquiring the road nodes is not possible, unlike the buildings. Each way has the ID number of every node that is a member of it. Therefore, it is possible to obtain every node required to make the road network by repeatedly requesting all missing nodes until no more exist.
Once the OSM road data is processed into a list of road objects, they are stored in an undirected graph. This is beneficial as it allows us to use graphing algorithms to explore the road network. This is done during the precompute pipeline as the initial construction of the graph can be quite expensive. Each OSM node is added to the graph and connected with an edge to its neighbouring nodes in the OSM way. As an OSM node can be in multiple ways this builds a full road network out of all points in the network. This creates a massive data structure and is too large to run algorithms on within the required frame rate; it must be compacted into a graph holding the same information using fewer nodes. Nodes with a degree of two are removed and the two edges incident to it are combined into one. Each edge contains useful information like the length of the road and a list of points that the road goes through. This means that each node in the new graph represents a junction on the road network and all the information is fully captured.

To build the roads in-game, a Bézier path is generated which

follows the edges of a path through the graph. This is done for all paths extracted from the graph such that each edge is only in a path once. These paths are greedily selected by picking the longest undrawn edge and expanding outward. Edges are only added if they are of the same road class such that A-roads are only connected to other A-roads so they can be textured differently. If there are multiple roads of the same class which could be grown from then the road with the smallest turn degree is picked.

These paths are converted into Bézier paths by converting the longitude-latitude coordinates of a node into a Vector3 and then creating a Bézier path from this list. This path is then transformed into a Vertex path and its mesh is created and triangulated by Sebastian Lague's PathCreator asset library [10].

With the roads represented as a graph, it is easy to run standard graph algorithms whilst the game is being played. After doing some user testing we implemented a restricted Dijkstra's algorithm for navigating the road network to help users get to the next mini-game location. This involved exploring all nodes within 10 edges of the closest node to the player. The path to the closest node to the next checkpoint is used to update the road chevrons to guide the player in the correct direction. This heuristic allows it to work in disconnected graphs (e.g. cities divided by rivers) and on large graphs without heavily affecting calculation time. Without such a well-defined structure this would have been a large undertaking but the already implemented road graph made it a simple addition to the game.

Terrain Generation

Unity's terrain system was used as it already supported the features that were needed 16. This included the terrain level of detail, which could be set up with neighbouring terrain tiles seamlessly. The material however was swapped out for a custom shader which although took away the ability to use Unity's instancing, we could add custom shader effects like translucency and water.

For the terrain elevation data, the OpenTopography API [28] was used to fetch height data from the NASADEM Global Digital Elevation Model [29] with a raster resolution of 30 meters. For the water, MapBox [30] API was used with a custom style to fetch a water mask which was then fed into our custom terrain shader to differentiate between land and water.

Unity's terrain has multiple restrictions on the heightmap such as:

$$w = h = 2^n + 1 \{n \in \mathbb{N}\}$$
$$elevation[x, y] \in [0, 1] \; where \; x \in [0..w], y \in [0..h]$$

To match these criteria, the fetched elevation is first normalised between its minimum and maximum height and then up-sampled with bilinear filtering, using a higher value of n for extra subdivisions and a smoother result. Upon implementing location selection, we encountered the problem of popular landmarks being on the corner of a tile and therefore not an ideal location for gameplay. To account for this, four tiles are chosen around the user's selected location and are stitched together on finishing the pipeline.

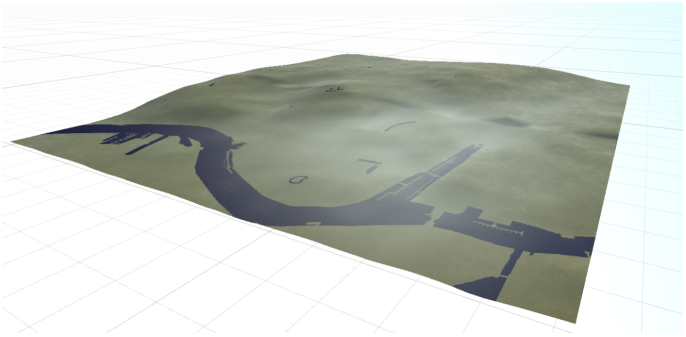

Figure 16: Buildings with multiple windows

Location Selection

We made two iterations of the location selection functionality for the users to decide where they want to play and added it to the UI. The first iteration used a model of the UK split into several chunks each representing a populated city in that region. The user would then select a city that would be loaded into the relevant location using the precomputed tiles. Callbacks were added for hovering over a region and enabling a pulsing shader to enhance visual feedback. This approach restricted the locations you could select to a few places in the UK. Furthermore, this approach prevented the user from starting wherever they wanted within the selected tiles. Overall this meant the number of locations and routes available were limited and would cause more repetitive gameplay which led us to overhaul this feature.

The second iteration (Figure 3) used the Mapbox static tiles API [30] to get a map of the world using a web request. The map would represent one tile centred at a given latitude and longitude at a certain zoom level. A translucent shader on a plane was used to identify which areas have been precomputed and are available to play on. The user can zoom in or out to decide which location they want to choose and place a marker to select their starting location. The chosen tile, as well as the three others closest to the marker, were loaded in for route generation. This method allows for varying routes and more playable regions throughout the world rather than just a few cities in the UK.

Building Mass Generation

Context Free Grammars Research into building mass generation yielded the idea of using context-free grammars and subdivisions to generate building facades. This has already been implemented in other procedural building generations [31]. Planning for this feature started with organising buildings into types that have similar facades.

Each facade then had a grammar assigned to it based on its building type. An example of a grammar is given for an office building and more can be found on the wiki page [2]:

```
Facade ::= Entrance Levels Roof
Entrance ::= Door | Door Porch | Porch
    ↪ Door
Door ::= "glass door" | "metal door" |
    ↪ "wooden door" | "revolving door"
    ↪ | "automatic door" | "sliding
    ↪ door"
Porch ::= "concrete porch" | "stone
    ↪ steps" | "ramp"
Levels ::= Window | Window Levels
Window ::= Large-window | Small-window
Large-window ::= "floor-to-ceiling
    ↪ window" | "bay window"
Small-window ::= "vertical window" | "
    ↪ horizontal window"
Roof ::= Flat-roof | Pitched-roof
Flat-roof ::= "flat roof" | "parapet
    ↪ wall"
Pitched-roof ::= "gable roof" | "hip
    ↪ roof"
```

An abstract recursive descent parse had to be implemented for each type of grammar that was used. The parser was used to verify the correctness of the grammar to which it had been assigned and to apply the assets to the building mesh.

Buildings were categorised into several distinct types and sorted by their prevalence. We used this to prioritise their development. For example, prisons are a rare building type so they weren't worth implementing on a project of this limited time scale. Grammars were assigned to buildings based on the tags that are received from the Overpass API. For buildings with no tags, we created a default grammar that created a generic building.

Applying assets to buildings was the most time consuming part of this task. When applying doors, it was necessary to check for the face closest to a road and that was not in contact with another building face (e.g. a terraced house). The application of windows was more challenging because the width of each face on a building was not homogeneous and varied between buildings. This meant that there had to be a calculation for the number of windows that could fit across each face and then given a grammar, a decision of how many windows to place on this face. This had to be repeated for the number of levels each building was calculated to have. An initial solution simply placed as many windows as possible on each face but this made every building look like office buildings. The last heuristic used was to place no more than 2 or 3 windows (Figure 17) per side of a building (there can be multiple faces per side) depending on the building size.

Window assets were created by first walking through the streets of Bristol, and taking photographs of windows. These images were imported into Blender as a plane and the faces were extruded to create a 3D asset that could be imported into Unity.

Figure 17: Buildings with multiple windows

We had a set of material, window and door assets. So that each museum produced by the game didn't look identical, a probability distribution was used for each grammar to apply these assets. The default distributions for each asset type were initialised with values that summed to one, this would be the distribution for the default grammar. When a different grammar had been assigned, then the distribution would be updated to make different assets more or less common. The discrete probability distribution P can be updated to obtain P' by setting the probability of asset a to $newProb$.

$$P'(a) = newProb$$

$$\forall n, n \neq a \quad P'(n) = P(n)\frac{(1 - newProb)}{1 - P(a)}$$

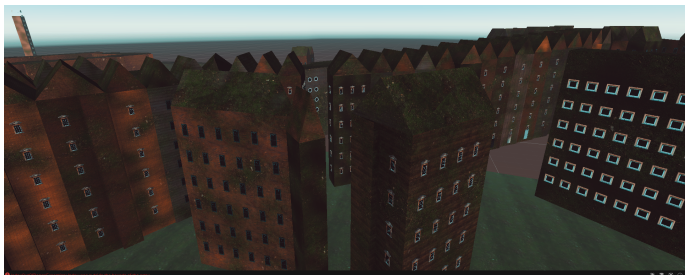

Figure 18: Buildings with multiple variations of windows and materials assigned with probabilities based on the building type.

To procedurally generate triangular prism roofs on buildings, we had to find the convex hull of the building footprint. We solved this using a heuristic method that obtains an approximate rectilinear convex hull instead. This is done by creating a bounding box based on maximum X and Z values around the building. The midpoints on two sides are extruded upwards to generate the triangles on each side of the prism. The prism then has its mesh triangulated and a material applied (Figure 19).

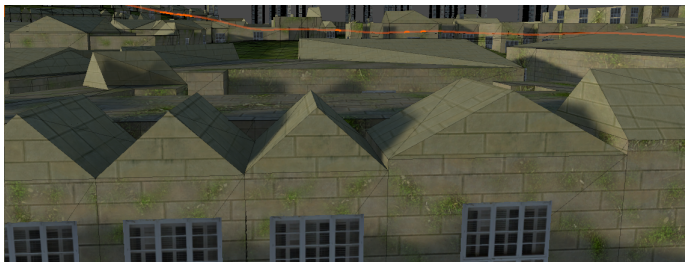

Figure 19: Picture of triangular roofs

Having evaluated the results of the Context Free Grammars approach, we determined that a more sophisticated solution would be required. The reasoning for this was twofold, firstly, adding new features to the grammars was time-consuming and manual often requiring large amounts of code. Secondly, we realised that despite having realistic textures and assets, a large component of photorealism is geometry. The system was incapable of small details like insetting windows into walls and creating more complex features. Whilst this would have been possible to implement, the time cost per feature was too high to justify hence we researched potential alternatives.

Generators Whilst researching, we came across an addon for Blender called Buildify [6], this tool used Blender's procedural mesh system called Geometry Nodes [32] to generate a building mesh given a 2D footprint. The addon worked by converting the footprint to a curve and then sampling points along that curve subject to a few constraints. At each sampled point, an asset of the appropriate type (wall, pillar, details) is instanced creating realistic buildings.

Having tested a default building (Figure 20) in Unity we decided this would be the best approach going forwards and began planning how to integrate this technology. We realised that we would need several variations of the Buildify generators to cover the spectrum of buildings we received data from (Figure 21). We allocated these generators across team members and George created a 32-minute tutorial [18] on how to create and modify them to suit our needs. Due to the level of abstraction provided by Buildify, most of the generators did not require modification of the underlying Geometry Nodes and were a case of creating 3D models to replace the default ones.

We also needed a way to run Blender from Unity and took advantage of Blender's headless mode to automatically launch the program and run the Geometry Nodes. This was accomplished using a Python script that was run when Blender opened. On the Unity side, a JSON file containing building footprints and additional meta-data was written to a directory accessible by both programs. The Python script would then read these footprints and create the relevant buildings. To improve the efficiency of the program, we created a duplicate of all Buildify assets in Unity meaning we could send transforms instead of models greatly reducing the space required and making it quicker.

Figure 20: An example of a Buildify building using our materials

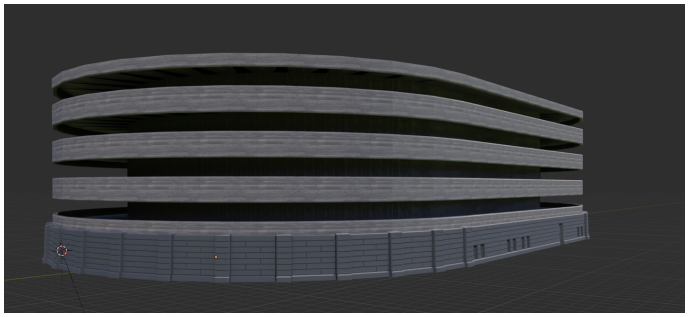

Figure 21: An example of a custom generator used for carparks

Once Blender had finished generating, the buildings were written to another shared JSON file and read back into Unity.

Optimisations

The main issue our optimisations tried to tackle was the number of batches. Batches are collections of geometry that can be rendered in one go without rebinding material data. Changing material has an overhead, so the fewer batches the better the performance. The first method to do this was splitting the world into a regular grid referred to as 'chunks'. A node in the pipeline takes meshes within these chunks and merges them all by material, reducing the overall batches.
An effort was also made to reuse as many materials as possible. For buildings, we had only a few preset materials which were shared. Other attempts were made to reduce instantiating new materials, for example, road textures were

initially scaled by creating a new material for each road and setting a parameter number of dashes, however, this creates a new batch for every single road. Instead, scaling the UVs of the road by its length meant all roads could share a material. The more materials that are shared, the more effectively they can be batched - either by merging by chunk or by Unity's batcher.

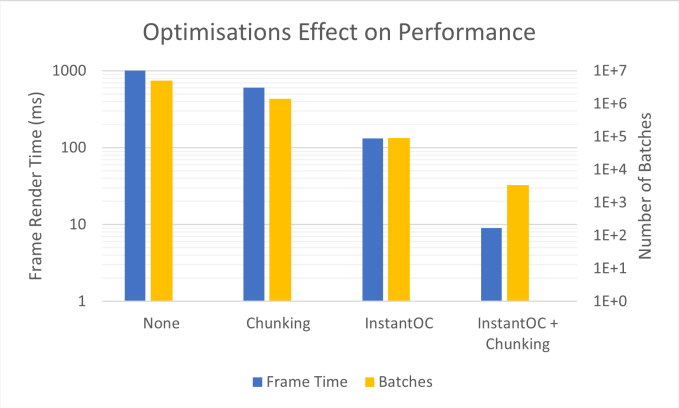

Figure 22: The effect on frame render time and batch size with different optimisations

Furthermore, we integrated InstantOC [33] dynamic occlusion culling into our game. We changed our pipeline and asset creation workflow to create the correct structure for this framework, giving each building part multiple levels of detail (LODs). InstantOC can be used for procedural worlds as no baked information is used, unlike Unity's occlusion culling solution which requires the scene to be static when built. By giving each chunk a collider and adding the IOClod component to it, that chunk can be checked for the correct LOD and occlusion by InstantOC via random raycasts from the camera each frame.

Another step considered we made to reduce the batch numbers was to dynamically create texture atlases in the pipeline to reduce the number of materials even further. Within a chunk, every material that shares a shader would have its textures collected and packed within a texture atlas before merging. The data to reference a particular texture in an atlas was then packed per vertex into two of the UV channels to be referenced by the shader. This meant chunks could be merged into one whole mesh if using the same shader, rather than one mesh per material. However, this optimisation was removed from the final implementation as the move to tri-planar mapping over UV mapping did not allow for atlas integration without our own tri-planar implementation.

Grass and Asset Scattering

As our world is designed to be overgrown, the denser the foliage the better. However, because of the large amount of geometry, overdraw, and memory used, we are limited by in-game performance.

For grass, initial optimisation took similar approaches to the methods outlined in the Optimisations section, with the grass being merged into larger chunks. The problem with this method however is the nature assets were largely repeated, and although merging the meshes reduced batches it also massively increased memory usage. Rather than storing positions and repeating the mesh, the merged meshes were all unique and therefore could not be shared. As the number of meshes rose into the millions, this memory usage was too large and we had to instead opt for an instancing approach.

The first instancing attempt sent grass instance data in batches directly from the CPU to be rendered. For instancing from the CPU, the maximum batch size is 1023 so the world was again split into chunks with batches being sent from the chunks in view. This instancing saw a huge improvement in memory (from consuming almost 10GB down to 10 megabytes) as positions were calculated when needed rather than continuously stored. The number of instances however was still a problem, if a million instances were shown then the number of batches would increase by almost one thousand. Furthermore, there is a performance overhead when sending large amounts of data from the CPU to the GPU, which could lock the rendering thread.

Our final implementation used indirect instancing (Figure 23). Instance and argument data are provided via compute buffers on the GPU so that there is no transfer of data between hardware other than control calls. The instance data is calculated for each frame by a custom compute shader, where data for each instance within that camera view is added to an append buffer, allowing for a variable number of instances. This count is copied over to the argument buffer and sent to be instanced. The bottleneck of sending data is removed as the GPU checks the same memory for the updated instance information every frame.

Due to the parallelisation of the GPU, hundreds of thousands of instances can be processed every frame with little overhead. However, checking the millions of instances within the whole map whether they should be occluded is still inefficient. The compute shader for grass, therefore, samples positions within a regular grid in front of the camera, and creates an instance for each grid cell given it is not culled. For each grid cell, the position is offset according to a clumping texture, set to the height of the terrain by sampling from the tile's heightmap, and given a random scale depending on another texture. All the randomness is deterministic with the seed being the cell's position, so the resulting transformation for each cell is constant.

The position is then transformed into the camera's projection space. If this projection is outside of a given range then the grass isn't in view of the camera, known as frustum culling. This projected position is then used to sample the camera's depth texture, which when compared against the projected z component can test whether another opaque

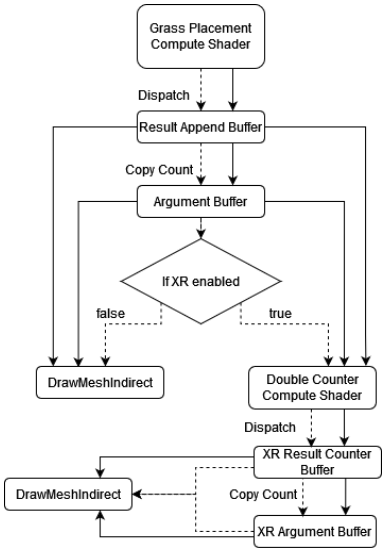

Figure 23: Indirect instancing architecture for grass, solid lines representing data flow on GPU and dotted representing CPU control flow.

object is blocking that instance, which is occlusion culling. As all these operations are mostly reading textures available on the GPU, therefore the culling process is efficient with the grass' compute shader taking only 0.049ms on the GPU when considering over 300,000 cells.

An effort was also made to place the sample on the grid so that the first cells appended were close to the camera, and the last cells the furthest from the camera, rather than the naive method of a fixed grid around the camera. Append buffers in compute shaders are random access, however, they are appended relatively synchronously (Figure 24). As no z-sorting is done with indirect instancing [34], it is important that opaque objects are generally rendered front-to-back to reduce overdraw[35]. If rendering grass in the worst case back-to-front, the render thread time averages 7.1ms, while with our sampling render order optimisation the render thread time averages 5.8ms, a 22% increase in overall rendering performance.

Random cells are also skipped as the grid gets further away from the camera to reduce density with distance. To give the illusion of the same density, as distance increases the width of the instance increases and the instance turns to face the camera, occupying a larger portion of the screen. These effects would be seen closeup however with distance go unnoticed.

The shader itself needs to be set up for instancing also. Given we were using URP and were limited to ShaderGraph, indirect instancing was unsupported and therefore had to be manually injected [36] with HLSL code to overwrite the object-to-world and world-to-object matrices to the ones defined in the instancing data. Furthermore, a variety of other techniques were used for visual interest.

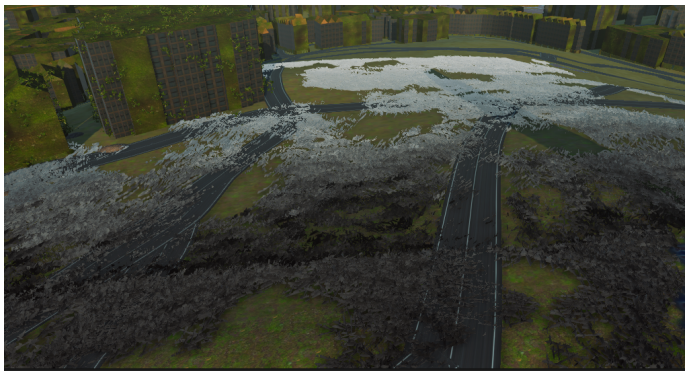

Figure 24: Render order of grass shown, black being first and white being last.

Translucent shading and procedural wind animation inspired by Crysis' vegetation [37] were used on all foliage. A compute shader generated a global wind texture using a multiple scrolling gradient noise, then used to offset foliage vertex positions according to a weight in the vertex's colour data, and translucency by considering the dot product between view direction and light direction offset by the normal[38]. Grass specifically has a large amount of aliasing at far distances due to the large overlap of alpha-clipped billboards, which was reduced by methods similar to the ones shown in the Ghost of Tsushima procedural grass GDC talk [39]. As the distance from the camera increases the smoothness of the grass decreases, the normal tilts to face toward the camera and there are fewer differences in ambient occlusion, making the grass more uniform and collected. To break up any uniformity, each grass instance samples from a colour texture to give colour variation and the UVs are transformed to a random texture on an atlas.

Other instanced meshes like the ivy on the buildings and the debris on the ground used a similar technique to the grass. The difference however is that the positions were calculated in the pipeline. Another chunking method was used, but each chunk stored its own compute shader. Each chunk is only responsible for storing and culling the instances within its bounds, all referencing the same output append buffer. On runtime, the nine chunks surrounding the camera were selected and their compute shaders dispatched, culling the stored instances in the same way described above. This allows for only nearby instances to be considered, and while separate compute shaders may increase the memory usage there will be a reduction in the overhead which would be caused by instead binding the correct compute buffers each frame. To help with z-sorting, the centre chunk the player is on is appended first, followed by the exterior chunks. The distance to the camera is calculated so further instances are added to another append buffer for a lower-poly level of detail model to reduce geometry.

Both XR and single screen needed to be supported for development reasons, however single pass stereo rendering [40] handles instancing data differently. The number of

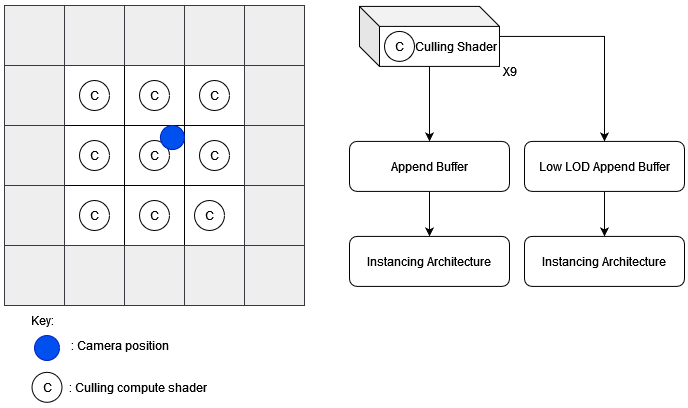

Figure 25: Diagram for general instance culling architecture. Instancing Architecture is similar to Figure 23.

instances is doubled and inter-weaved [41], one for each eye. The solution tested for this was if XR is currently in use, use a counter buffer of constant size and run a compute shader which increments the counter twice for each element in the append buffer and writes that element to the counter buffer. This accounts for the double count without actually doubling the number of instances. For the frustum and occlusion culling in XR, only one eye was considered.

The last significant tech for instancing was the creation of density maps. The final instance positions are sampled against the density map representing the probability of an instance being there. This will always be 0 where there is a building or water. The mask involved simply rendering a camera in an orthographic bird's-eye view to get a building mask, applying a threshold and combining it with the water mask (See Terrain Generation). However, for a variety in asset distribution, a distance field compute shader was created to vary density depending on the distance to a building. Our implementation took a wave-propagation approach, which when run iteratively 'diffuses' the mask outward at a constant rate. The output on an example mask (Figure 26). These results were combined with random gradient noise to give a more varied density map.

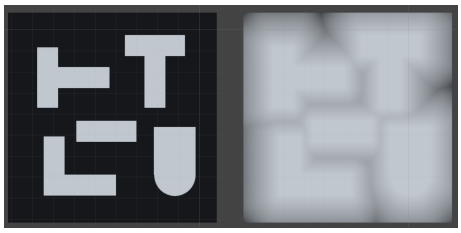

Figure 26: Example mask and its approximate distance transform after wave-propagation.

Technical Art

Visual effects and materials play a significant role in game development by making games more engaging and immersive. We regularly took advantage of Unity's VFX Graph and Shadergraph, user-friendly tools to improve the look and feel of the game as well as to assist gameplay.

Visual Effects The visual effects of the game mostly consist of particle systems, they provided visual indicators to the player and also ensured that no area of the game felt boring. Whilst some of our particle effects were modifications of assets created by others, some were bespoke and created to serve a specific purpose. The main example of this is the target shooting game in which the arena has several particle effects. We used Unity's Visual Effect Graph to create these systems by spawning particles within a set of conditions and then updating them each timestep using another set of conditions. This was all handled using a node editor which provided an intuitive means to create complex effects.

Real-time Water To represent water we designed a shader that imitated movement by using a time scale to continuously offset two normal maps and combine them. We decided on this approach after weighing up different methods such as implementing Gerstner waves [42] which would have provided an exact solution of the Euler equations for periodic surface gravity waves which would provide us with realistic waves. However, this method would require adding vertices and displacing as described by the equations. This would be expensive and not a key feature of the game. Alternatively, normal mapping provided the desired level of detail without significant overhead (see video [21] - 2:49).

Cracked roads Shader The game's setting is in the future where the roads wouldn't have been maintained. To replicate this we added some cracks to the roads using a shader. We did this using Perlin noise to scatter a second normal map and interpolated between them to emphasise the damage. Once again this was a cheap and effective method for producing the effect we wanted (see video [21] - 1:02).

Vignette Motion sickness is a common occurrence in virtual reality games due to the user's eyes registering simulated movements but being static in real life. This is mainly true for users who have not had much experience with virtual reality and therefore have not become accustomed to it. Our movement system is susceptible to triggering this and to counter it we added a vignette. This restricts the user's field of view and is a commonly used technique in the industry [43]. The vignette was taken from the Unity XR Interaction Toolkit [44] and was converted to work with our render pipeline. We than added a controller to make it toggleable (see video [21] - 1:23) so that experienced users can still be fully immersed in the gameplay.

Player X-Ray Vision Given that we have an open-world game it can be easy to lose sight of the other player. This is detrimental to the social gaming aspect that is present in multiplayer games. Our solution to this was to add

another pass to our renderer and created a simple material to represent the silhouette of the player. Any obscured regions of the player were rendered last which allows users to constantly keep track of their opponent (see video [21] at 1:30.

Parallax Windows At a point in the project, we ascertained that to further improve the realism of the game, we would need to have interiors to buildings. Given our restrictive budget for graphical improvements, this needed to be performed in an efficient manner that had little overhead. After some research, we came across a type of shader that uses the camera's view direction to project a cube map 27 behind a window creating the illusion of a 3d interior (see video [21] - 2:40). Following a tutorial [45] we were able to create this shader within Unity and generate interiors for buildings with little performance cost.

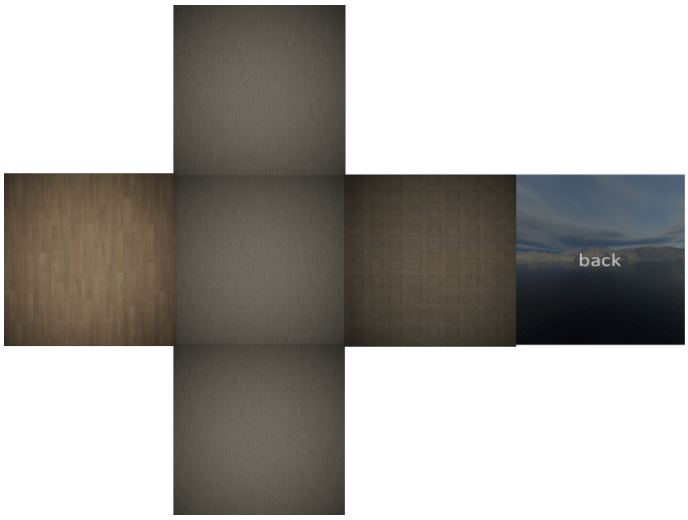

Figure 27: An example of a cubemap [46] used in our game.

Networking

Our game requires networking to support two players who will be connected P2P, with one acting as a host to run game server operations. One extra factor we needed to take into account is that to avoid port-forwarding and firewall issues there needs to be an intermediary relay server. A relay server is where all traffic is routed through but does not process any game logic. To implement this we needed to choose a library framework which handles all our needs. We were advised that Photon Unity Networking (PUN) is a good option but we also looked at alternatives, namely Netcode for GameObjects (NGO). Both solutions would fulfill our requirements, are similar in the features they have and are similarly implemented. However, we decided to use NGO for a variety of subtle reasons: it integrates with Unity Gaming Services (UGS) meaning that no third-party cloud providers are required; it supports fast development iteration as it implements functionality by adding components to GameObjects; and it is purpose-built for Unity keeping more of the technology stack within the same ecosystem.

Figure 28 shows the overall architecture of the game with two players connected through a UGS Relay server where one player acts as the host for the game server. Additionally, networking was reworked to be built upon an FSM to manage the handling of network states. Each state is a network handler for a specific connection state which inherits from the same parent class, allowing them to be seamlessly swapped in and out of the connection manager. This was adapted from Unity's networking example game called Boss Room[19]. It also includes handling NGO authentication and session management.

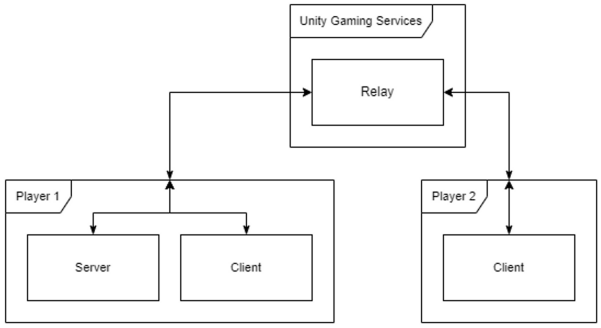

Figure 28: A diagram showing how NGO has been used to create a 2-player game environment.

Voice Chat

One key insight found during user testing was that we needed to increase player interaction and improve communication. We utilised voice chat as one of the tools to achieve this goal. One advantage of using UGS is that it offers a voice chat service, Vivox [15], which can be set up to easily integrate with our existing networking through NGO, making integration straightforward and immediately robust. Following this update, we found that you felt much more involved with the other player being able to hear them clearly even with lots of background noise.

References

[1] V. Driessen, "A successful git branching model," 2010. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/.

[2] NPC Multi-storey Car Park, *Team b wiki*, old url: https://github.com/NPCMS/ElementalExplorers/wiki, Notion, 2023. [Online]. Available: https://glow-maiasaura-ec6.notion.site/Wikis-Elemental-Explorers-18d0f3bbabe0461b8fd24d868a0a4c86.

[3] T. ICHÉ, *Now available: The spaceship demo project using vfx graph and high-definition render pipeline*, Aug. 2019. [Online]. Available: https://blog.unity.com/technology/now-available-the-spaceship-demo-project-using-vfx-graph-and-high-definition-render.

[4] A. Hunt, *The pragmatic programmer : from journeyman to master*. Addison-Wesley, 2000.

[5] T. B, *Minigame ideas*, Mar. 2023. [Online]. Available: https://docs.google.com/document/d/1lMSH33FpcNOF_w3s40eY8pS0pzOVP9f_GA9ZbCHMn-8/edit?usp=sharing.

[6] P. Oliva, *Buildify 1.0*, 2022. [Online]. Available: https://paveloliva.gumroad.com/l/buildify.

[7] *Megascans: The largest and fastest growing 3d scan library*. [Online]. Available: https://quixel.com/megascans.

[8] *Monday.com*, 2012. [Online]. Available: https://monday.com/.

[9] L. MILGRAM, "Law of diminishing returns," *Managing Smart*, pp. 51–51, 1999. DOI: 10.1016/b978-0-88415-752-6.50044-8.

[10] S. Lague, *Path creator*, https://github.com/SebLague/Path-Creator, 2022.

[11] Prenominal B.V., *Realtime csg*. [Online]. Available: https://realtimecsg.com/.

[12] K. Henkel, *Parabox csg*. [Online]. Available: https://github.com/karl-/pb_CSG.

[13] Unity Technologies, *Probuilder*. [Online]. Available: https://unity.com/features/probuilder.

[14] Valve, *The lab*. [Online]. Available: https://store.steampowered.com/app/450390/The_Lab/.

[15] Unity Technologies, *In-game voice and text chat (vivox)*. [Online]. Available: https://unity.com/products/vivox.

[16] *Github super liner*, 2020. [Online]. Available: https://github.com/marketplace/actions/super-linter.

[17] *Blender 3.5 python api documentation - blender python api*, 2023. [Online]. Available: https://docs.blender.org/api/current/index.html.

[18] G. Wigley, *Generator tutorial*, https://www.youtube.com/watch?v=kKlFrX6IIaw, 2023.

[19] Unity Technologies, *Boss room architecture*, 2023. [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/learn/bossroom/bossroom-architecture.

[20] *Meat hook*, Mar. 2020. [Online]. Available: https://doom.fandom.com/wiki/Meat_Hook.

[21] E. Explorers, *Elemental explorers video*, May 2023. [Online]. Available: https://www.youtube.com/watch?v=2H-Qpqbo92M.

[22] G. K. Mark, D. Alan, and A. Jen, *Affective videogames and modes of affective gaming: Assist me, challenge me, emote me*, 2005. [Online]. Available: http://www.digra.org/wp-content/uploads/digital-library/06278.55257.pdf.

[23] Unity Technology, *Editing curves*, 2021. [Online]. Available: https://docs.unity3d.com/Manual/EditingCurves.html.

[24] Siccity, *Xnode*, https://github.com/Siccity/xNode, 2022.

[25] P. Projects, *Flask*. [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/.

[26] David Eberly, *Triangulation by ear clipping*. [Online]. Available: https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf.

[27] Sebastian Lague, "Ear clipping triangulation," [Online]. Available: https://github.com/SebLague/Ear-Clipping-Triangulation.

[28] OpenTopography, *Opentopography*, 2023. [Online]. Available: https://opentopography.org/.

[29] NASA JPL, Distributed by OpenTopography, *Nasadem merged dem global 1 arc second v001*, 2023. [Online]. Available: https://doi.org/10.5069/G93T9FD9/.

[30] MapBox, *Mapbox*, 2023. [Online]. Available: https://www.mapbox.com/.

[31] G. Nishida, A. Bousseau, and D. G. Aliaga, "Procedural modeling of a building from a single image," *Computer Graphics Forum*, vol. 37, no. 2, pp. 415–429, 2018. DOI: https://doi.org/10.1111/cgf.13372. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13372. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13372.

[32] *Geometry node*, May 2023. [Online]. Available: https://docs.blender.org/manual/en/latest/render/shader_nodes/input/geometry.html.

[33] frenchfaso, *Instantoc*, 2023. [Online]. Available: https://assetstore.unity.com/packages/tools/camera/instantoc-dynamic-occlusion-culling-lod-6391#description.

[34] Unity Technology, *Graphics.drawmeshinstancedindirect documentation*, 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html.

[35] Unity Technology, *Opaquesortmode documentation*, 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rendering.OpaqueSortMode.html.

[36] echu33, "Shadergraph injection similar to this answer," 2023. [Online]. Available: https://forum.unity.com/threads/shadergraph-instancing.1339943/.

[37] Tiago Sousa, Crytek, "Vegetation procedural animation and shading in crysis," 2023. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-iii-rendering/chapter-16-vegetation-procedural-animation-and-shading-crysis.

[38] Alan Zucconi, "Fast subsurface scattering in unity," 2023. [Online]. Available: https://www.alanzucconi.com/2017/08/30/fast-subsurface-scattering-1/.

[39] Eric Wohllaib - Sucker Punch Productions, *Procedural grass in 'ghost of tsushima'*, 2023. [Online]. Available: https://www.youtube.com/watch?v=Ibe1JBF5i5Y.

[40] Unity Technology, *Single pass stereo rendering documentation*, 2023. [Online]. Available: https://docs.unity3d.com/Manual/SinglePassStereoRendering.html.

[41] Unity Technology, *Single pass instanced rendering, procedural geometry*, 2023. [Online]. Available: https://docs.unity3d.com/Manual/SinglePassInstancing.html.

[42] NVIDIA, "Chapter 1. effective water simulation from physical models," [Online]. Available: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models.

[43] D. Murphy, "Oculus wants to end vr motion sickness — here is the solution," 2021. [Online]. Available: https://mixed-news.com/en/this-pc-tool-helps-prevent-vr-motion-sickness/.

[44] U. Technologies, 2023. [Online]. Available: https://github.com/Unity-Technologies/XR-Interaction-Toolkit-Examples/tree/main/Assets/Samples/XR%5C%20Interaction%5C%20Toolkit/2.3.0/Tunneling%5C%20Vignette.

[45] *Fake interior effect in unity using shader graph*, May 2020. [Online]. Available: https://www.youtube.com/watch?v=rC4BHR-Cx0s.

[46] A. Creations, *Fake interiors free: Vfx shaders*, Apr. 2021. [Online]. Available: https://assetstore.unity.com/packages/vfx/shaders/fake-interiors-free-104029.