University of BRISTOL

Shifting Sands

COMS30043 Team Project - Seal Team 7

Peter Lillistone
Project Manager

Harry Greentree
Lead Programmer

Finn Cooper
Developer

Bhagavath Achani
Lead Designer

Sergi Lange-Soler
Developer

Josh Chatten
Developer

Contents

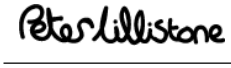
1 Signed Declaration

We hereby declare that the work presented in this report has been completed solely by the team members listed below. We confirm that we have each contributed to all deliverables, have read the report in full, and agree with its contents.

During the development process, we used the following AI tools:

- **Coding Assistants** – Cascade, Cursor, GitHub Copilot, Gemini Code Assist and Rider Auto-Completion. Used to provide context-aware code suggestions, such as boiler-plate code suggestions, significantly accelerating routine coding tasks and reducing development time.
- **Chatbots** – ChatGPT and Claude. Used primarily as an interactive debugging assistant, helping interpret bugs and error messages, brainstorm ideas for resolving issues, and explore alternative solutions to resolve complex coding challenges.

Team Members and Signatures:

Peter Lillistone: 

Harry Greentree: 

Finn Cooper: 

Bhagavath Achani: 

Sergi Lange-Soler: 

Josh Chatten: 

Fig. 1. Photo taken on Games Day.

2 Top Five Contributions

1) **Real-time, Responsive Terrain Generation:** Developed a system that uses the Kinect's depth data to capture changes in the sand topography, to create a seamless real-time mapping between the physical sandbox and the digital terrain. This enables a highly responsive and strategic input method where players can actively terraform the battlefield and influence gameplay through intuitive, tactile interaction.
2) **Image Processing and Masking:** Reduced random noise in the depth image to the point of being imperceptible, using low-pass filtering and the exponential moving average, with α selected adaptively according to pixel stability. Developed a novel method to identify the user's hands, arm and head in the depth image, using a combination of thresholding, dilation, vector arithmetic and ML-driven computer vision.
3) **Hand Tracking and Reconstruction:** Implemented hand tracking by leveraging Google's MediaPipe library within a dedicated Python process. This process analyses the Kinect's colour image to detect 3D hand landmarks. These landmarks are then streamed back to the Unity application, enabling accurate hand reconstruction in-game and precise lower arm masking.
4) **Custom Enemy Implementation:** Created a diverse roster of enemy types using a finite-state-machine approach and object-oriented principles such as inheritance and polymorphism to allow for a wide range of strategies to be used in-game. Each enemy type has a physics script to enhance or, in some cases, supersede the Unity Rigidbody physics. As well as this, enemies use a version of the A* algorithm to path-find around different terrain.
5) **Fully In-house Visuals and Art Style:** A unique toon-style visual identity created through 20+ custom 3D models, 25+ fully bespoke UI assets, hand-crafted particle systems, VFX, custom toon-shaders, and post-processing effects.

3 Team Video, Website & GitHub

You can view our trailer at the following URL:
https://www.youtube.com/watch?v=Qr2ObJGpmOg

You can view our technical video at the following URL:
https://www.youtube.com/watch?v=LWqZfMd8NSc

You can view our website at the following URL:
https://sealteam7-4976a.web.app/

You can view our GitHub repository at the following URL:
https://github.com/lm22433/2024-SealTeam7

4 Nine Aspects

4.1 Team Process

- Weekly in-person meetings at the start of the week to coordinate plans for the week. Including assigning issues from the Kanban board, breaking down tasks and documenting bugs. (See Section 6.2)
- Weekly in-person code review and merging sessions.
- Used GitHub's Kanban board to manage tasks and track progress. (See Section 6.7.1)
- Held sprint retrospective meetings after each milestone to evaluate progress and identify areas of improvement. (See Section 6.2)
- Followed an iterative development cycle to rapidly prototype and refine gameplay features. (See Section 6.2)
- Applied test-driven development to improve gameplay based on user feedback. (See Section 8.9)
- Engaged in regular pair-programming sessions to share knowledge and improve code quality. (See Section 8.8.3)
- Organised efficient ways of transporting large quantities of sand and other equipment between Queen's building and Merchant Venturers building. (See Section 6.6)
- Organised and evenly split "testing time" between members for usage of the Kinect and sandbox due to having access to only one Kinect. (See Section 6.6)

4.2 Technical Understanding

- Researched and integrated Microsoft Azure Kinect SDK for depth sensing and colour image capture.
- Investigated Google MediaPipe's hand landmark tracking pipeline for robust and low-latency hand tracking. (See Section 9.5)
- Explored and implemented shared memory inter-process communication using Win32 and POSIX APIs for Unity–Python integration. (See Section 9.1)
- Developed a semaphore-based synchronisation mechanism to maintain real-time Kinect data transfer between Unity and Python. (See Section 9.6.1)
- Evaluated various inter-process communication models—including ZeroMQ, POSIX IPC, Win32 API, and sockets—before selecting a low-overhead shared memory approach. (See Section 9.6.1)
- Developed an image processing pipeline including techniques such as low-pass filtering and the exponential moving average to smooth noisy Kinect depth data.
- Imposed a maximum height constraint on terrain mesh updates to reduce sensor-induced noise and ensure consistent gameplay interactions.
- Designed and implemented custom A* pathfinding for Enemy AI that dynamically adapts to terrain deformation and player manipulation. (See Section 9.7.2)
- Conducted performance analysis to balance depth sensing, image processing, and gameplay logic at interactive frame rates.
- Explored running in different operating systems and attempting to compile custom versions of packages to get around package incompatibilities. (See Section 9.6.3)

4.3 Flagship Technologies Delivered

- Real-time terrain manipulation via Kinect input with accurate topographical mapping.
- This required multiple layers of optimisation as well as image resizing and reorientation to translate into a digital world.
- A bespoke image processing pipeline, including noise reduction and hand masking, to deliver accurate and aesthetically pleasing gameplay.
- Full interactivity between physical sandbox and digital gameplay.
- Designed a fun and immersive game around using this novel interaction method.

4.4 Implementation & Software

- Used the Unity game engine to create the core game logic, visuals, and physics.
- A separate Python process handled Kinect data and processed depth information.
- Inter-process communication was managed via shared memory using the Win32 API or POSIX IPC. (See Section 9.6.1)
- Kinect images were captured in Unity and sent to Python, synchronised using semaphores.
- Implemented a custom damage model for both the Godly Hands and the Godly Core.
- Developed a score system based on enemy type, time survived, and combo streaks.
- Implemented a custom enemy spawning system.
- Implemented many unique enemy types using techniques such as inheritance and polymorphism. (See Section 9.7)
- Implemented projectile-based weapons for enemies. (See Section 9.7.3)
- Implemented an enemy and projectile pooling system to lower resource overheads. (See Section 9.7)
- Created Scripts used to enhance or supersede the built-in Rigidbody physics in Unity. (See Section 9.7.1)

4.5 Tools, Development & Testing

- Unity (C#) for core game development. (See Section 8.2)
- EmguCV (OpenCV C# wrapper) in Unity for terrain and depth generation. (See Section 8.2.2)
- Python + MediaPipe and OpenCV for hand landmark detection and tracking. (See Section 8.3)
- Shared memory communication using Win32 API and POSIX IPC. (See Section 9.6.1)
- Semaphore-based synchronisation of Kinect image data between Unity and Python. (See Section 9.6.1)
- GitHub for version control and project management. (See Section 8.1)
- Manual and user testing sessions to validate gameplay mechanics and tracking accuracy. (See Section 8.9)

4.6 Game Playability

- Simple and natural control scheme with a broad range of nuance created by having only the sandbox as an input method.

- Progressive wave-based enemy system with increasing difficulty.
- Multiple enemy types with varied abilities and strategies.
- Interactive terrain-based combat including burying, building and strategy.
- Real-time response between player actions and in-game outcomes.
- Experimented with finding the best setup to minimise physical and mental barriers between imagining how your hands were impacting the game world.
- Fully custom scoring system encouraging player engagement and mastery.
- Tutorial mode that gradually introduces game mechanics and enemy types.
- Visual and audio feedback systems that communicate game state and enemy threats clearly.
- Tooltip pop-ups that help inform players when enemy types first appear.
- Difficulty perfectly balanced for all skill ranges based on countless tests and score results. (See Section 8.9)

4.7 Look & Feel

- Fully custom toon shader for a stylised visual aesthetic. (See Section 9.8.1)
- Created approximately 20 hand-crafted 3D models. (See Section 3)
- Created 4 beautiful visual effects from the ground up using VFX graph. (See Section 9.8.2)
- Fully custom UI assets consistent with game theme.
- Distinct visual feedback for enemy types and damage states.
- Altered enemy physics to enhance user feeling when burying enemies. (See Section 9.7.1)
- Real-time shadows enhancing depth perception in the game world.
- Tested multiple types of sand to get the best feeling for players. (See Section 8.9)
- Tested and found the best locations for the tower and enemy spawns for better accessibility for players with different heights and skill sets.
- Designed and created a sandbox and surrounding play area to match the theme and fully immerse players using custom LED lighting and speakers with spatial sound. (See Section 8.7)
- Created a secondary waiting area for people not currently playing with news footage, music and posters introducing game elements. (See Section 9.8.3)
- Created a website to host a leaderboard and bonus information for players to read through. (See Section 3)

4.8 Uniqueness & Innovation

- Seamlessly blended physical interaction through the Azure Kinect v3 with a digital game environment, creating a tangible-digital hybrid gameplay experience.
- Unique “Godly Hands” mechanic allowing players to sculpt, destroy, and manipulate terrain in real-time using natural hand movements.
- Developed a novel terrain interaction model, where enemies adapt to terrain deformations generated by the player mid-battle.
- Novel interaction also leads to novel enemy types and ways to combat those enemy types.
- Designed the game to be played without any controller—purely using hand movements—making it highly accessible and immersive.
- Integrated a real-time feedback loop between depth sensor input and terrain updates for low-latency responsiveness.
- Innovation lies not just in gameplay mechanics, but in the architecture that connects heterogeneous systems (Unity + Python) through low-level shared memory IPC.
- Created a unique experience that brings memories of playing in sandboxes or on a beach back whilst generating new memories within a fun and immersive game.
- Created a unique experience for people who are waiting to play by having two rooms where you can either watch the current player or see the world through a news broadcast in the other room. (See Section 9.8.3)
- Created the whole experience thoughtfully and made sure that each aspect has purpose and intent to truly enhance the endless possibilities and imagination possible from a sandbox.
- Smaller innovations were also made to create a more accessible game, such as sound design elements, visual cues, tooltips and separate game modes, such as sandbox mode or endless mode.

4.9 Report & Documentation

- Created a comprehensive 30+ page technical report exceeding 15,000 words.
- Created a website with information and a leaderboard for players during Games day. (See Section 3)
- Created posters to guide the public and introduce game elements.
- Included detailed system architecture diagrams, gameplay mechanics, and design decisions.
- Provided performance evaluations and testing methodologies.
- Organised scripts accordingly and modularised them using industry standard abstraction techniques common in object-oriented languages, such as inheritance and polymorphism, to preserve readability.
- Maintained detailed GitHub documentation for developers and future work. (See Section 3)
- Kept an archive branch on GitHub from before the pivot. (See Section 3)

5 Abstract

Shifting Sands is an augmented reality (AR) game that merges the fun and exciting gameplay of tower defence games with the nostalgic feeling of playing in a sandbox. This is bundled together in an inclusive and immersive experience that is accessible to people of all ages and gaming experience, all to create the first ever sandbox-controlled video game. Across only 12 weeks of iterative development and exhaustive user testing, we have presented a fun and detailed concept of what such a game could look like and, in turn, laid the groundwork for future development into such a novel input mechanic.

Setting out, we established goals for the project that have helped drive development and focus the vision of the final result. The first of these is *sandbox integration*; at the core of everything should be the sandbox, and its interaction should feed back into every aspect of the game. We want to encourage active use of the sand, whilst keeping it intuitive. We also focused on accessibility; our aim was that the bar to entry should be as low as possible, allowing for anyone to enjoy and experience the game whilst still including nuanced systems and difficulty for those who have more experience or want a challenge. Finally, we aimed for the novelty, as we know that it is likely a one of experience, we really wanted to emphasise the interesting and novel nature of the technology.

5.1 Concept

The game's core concept is simple yet innovative: the player takes on the role of a godlike figure in a large-scale battle, with the goal being to defend their central Godly Core from waves of oncoming enemies by physically manipulating the sand to control the in-game terrain. This intuitive control system is powered by AR Sandbox technology, using the Azure Kinect's depth sensing capabilities to map the sand's topography to the in-game terrain. While AR sandboxes are commonly used in tactile learning settings like geoscience education [1], Shifting Sands reimagines and expands on this technology to deliver a uniquely interactive and immersive gaming experience.

5.2 Gameplay

Gameplay revolves around surviving increasingly difficult waves of enemies that spawn around the map and advance towards the Godly Core. The player uses their Godly Hands to dynamically alter the sand terrain. This allows for diverse defensive and offensive strategies: building mountains to block or redirect enemies, digging pits to trap them, burying/crushing foes under the sand and even flinging enemies across the map.

The challenge intensifies with the introduction of varied enemy types demanding specific tactics: flying units must be intercepted by rapidly raising mountains into their flight path; digging units tunnel beneath the surface, requiring the player to dig them out; parachutists that ignore fall damage, necessitating other methods like burial; and resilient heavy units, such as tanks, that may need to be buried multiple times to be defeated. This diverse roster demands adaptive strategies and skilful and intentional manipulation of the terrain.

Both the player's Godly Hands and the central Godly Core are susceptible to damage from enemy attacks and share a single health pool, compelling the player to protect both. Although enemies prioritise attacking the more vulnerable core, they will target the hands if they are within range.

Success is measured via a custom scoring system designed to reward skilful play, factoring in points awarded for different types of enemies killed, the total time survived, the player's health remaining at the end of a session, and bonuses for maintaining no-damage streaks. The ultimate goal is to achieve the highest score possible and top the leaderboard.

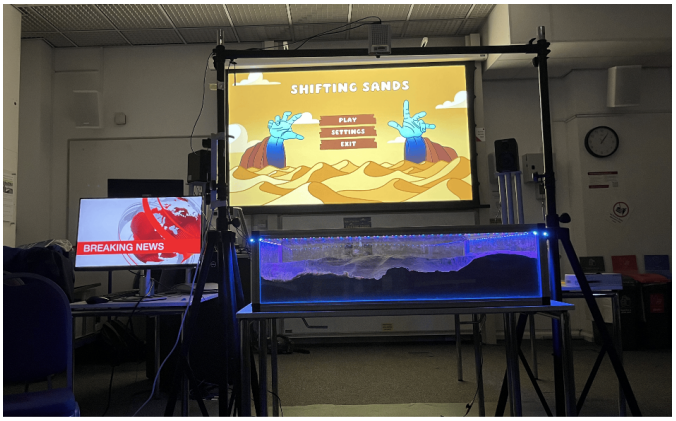


Fig. 2. Final setup on games day.

6 Team Process & Project Planning

6.1 Team Building

Although the team was established externally to the module, not all members were well-acquainted. To solve this, we took a number of steps:

1) We participated in the annual Game Jam organised by the Computer Science Society (CSS). This 24-hour event, centred around rapid game development in Unity, served a crucial dual purpose: it significantly boosted our team's practical skills and confidence with the Unity engine, while simultaneously acting as an effective ice-breaker that helped forge strong initial working relationships, culminating in our project winning the People's Choice award.
2) We also prioritised informal social interactions outside of the typical work setting. Regular, organised social gatherings allowed members to connect more personally, further reinforcing team spirit.

These activities contributed significantly to the supportive and collaborative atmosphere that was maintained throughout the project.

Fig. 3. Picture of Spooky Seal Team 7 winning CSS Game Jam 2024

6.2 Agile Workflow

Using agile workflow methodology, we worked in week-long sprints. To aid this, we aimed to have the majority of the team on campus five days a week, which strengthened our agile process by enabling more spontaneous, informal meetings and fostering a strong environment of inter-team feedback on code and design choices.

Each sprint would begin on Monday, with an hour standup meeting to discuss the project's current progress and create issues for the upcoming sprint. These also acted as brainstorming sessions for new features and helped to focus the long-term oversight of the project. Tuesday's meeting would then be used as a merging session, which allowed for the consolidation of PR reviews and in-depth testing before being accepted into dev.

Finally, Thursday's session would be used to create a demo build for our weekly review meetings. These demonstrations allowed us to get "client" feedback on the week's progress and bring that forward to the next sprint, helping create a highly adaptive workflow.

Over the course of the project, two sprint retrospectives were conducted, each following a major release. These retrospectives aimed to evaluate the development process critically, identifying successful practices and areas requiring improvement. A key insight gained was the importance of inclusive discussion regarding critical project aspects. Initially, private communication channels had been implemented for the digital and physical sub-teams to minimise information irrelevant to their specific tasks. In practice, this structure reduced inter-team transparency, complicated project management efforts, and led to the repetition of discussions. This issue was addressed by publicising all communication channels, thereby improving the team's transparency and collaboration.

6.3 Shifting Focus

Adaptability in our vision for the game has been key throughout development, with little precedence for what a game controlled entirely by sand would actually manifest as, it has required iterative and often significant changes to the core mechanics of the game. The largest of these occurred after the MVP release, in which we demonstrated potential in our underlying technology, but struggled to satisfactorily tie it together into a complete experience. This resulted in a rethink of the whole project, with a focus on emphasising the key technologies whilst abstracting out anything that distracted from the core functionality.

This new vision removed the multiplayer First Person Shooter (FPS) aspects in favour of shifting perspective back to the sandbox; from this point forward, the only input method for the player would be the sand. Through this simple medium, we would build up game mechanics to incentivise complex interactions, whilst still leaving room for players to experiment. The aim was to present a unique experience that demonstrated the potential of the technology.

6.4 Communication

Team communication took three primary forms; first, a Discord server, which allowed for structured, topic-specific conversations about the project and more long-term planning. The secondary form of communication was WhatsApp, which served as a good tool for short-term reminders and more time-sensitive discussions. Finally, the most essential form of communication was regular in-person meetings and collaborative work in the lab.

6.5 Managing Conflict

Throughout the project, the group maintained a collaborative and harmonious environment, free from major interpersonal or task-related conflicts. This was achieved through proactive, open, and honest communication, ensuring all team members felt comfortable voicing opinions, questions, and concerns. Key project milestones, such as sprint retrospectives, provided structured opportunities for reflection on successes and areas for improvement. To address potential disagreements, the team implemented a simple veto system, allowing each member one opportunity to halt a decision they strongly opposed. Although never used, its presence reinforced that every perspective was valued.

6.6 Logistics

Another aspect of the project that has proved difficult is managing the logistics, whether this has been getting equipment, finding spaces to develop in, or filling out paperwork. This has proven to be a considerable time sink in the project, with

a lot of going back and forth with the department to sort out blocking issues. Examples of these are as follows:

- Requirements for 3 separate room risk assessments to cover the BIG lab, 2.56 and 1.11.
- Approximately 5 requisition forms to order equipment for the project.
- Team undergoing lab and equipment inductions.
- Moving the setup between Queen's and MVB multiple times for panel days and mid-project room change.
- On average, weekly meetings with the tech team to discuss equipment and get technical advice.
- Outside of Games Day, having fewer than 10 hours of access to the actual room we were using for the final demo.

Due to the physical nature of our project, a lot of development revolved around testing on our rig, which posed challenges as we only had one Kinect and setup. This resulted in situations where work was bottlenecked by individuals being forced to wait to use the rig; to reduce this as an issue, two solutions were implemented. The first was creating a test environment that used procedural noise as a substitute for the Kinect's output (See Section 9.2.1); this allowed for rudimentary testing of features without needing access to the rig. Once features worked in the test environment, they were allocated testing time on the rig, maximising effective usage of the system and ensuring fair distribution between the team.

Fig. 4. Team setting up testing box and rig for MVP panel

6.7 Project Management

6.7.1 Kanban Board & Issues

To organise the project effectively, we utilised GitHub's Kanban board by breaking down the overall workload into issues. Each issue was then assigned to an individual team member, often considering their skills, current workload, and preferences, ensuring clear ownership. Our aim was to make each issue a self-contained unit of work by providing in-depth descriptions that clearly outlined the task requirements and relevant context. Team members were encouraged to use the issue comments to record progress updates, ask questions, discuss potential solutions, or note down relevant findings during development. This practice fostered transparency and created a record for each task. To further enhance organisation and visibility, we employed several key GitHub features:

- **Labels:** We used labels extensively to categorise issues based on priority and feature. This labelling system allowed us to quickly filter the board, understand the distribution of work, and identify related tasks easily.
- **Milestones:** GitHub Milestones were used to group issues related to larger objectives, often corresponding to our weekly sprint goals or major project deadlines like the MVP, Beta, and Final release builds.

This combination of a well-structured Kanban board, with detailed issues, labels, and milestones, streamlined project management and improved team coordination. Clear task ownership, open communication, and organised tracking helped us stay on schedule and maintain steady progress towards key project goals.

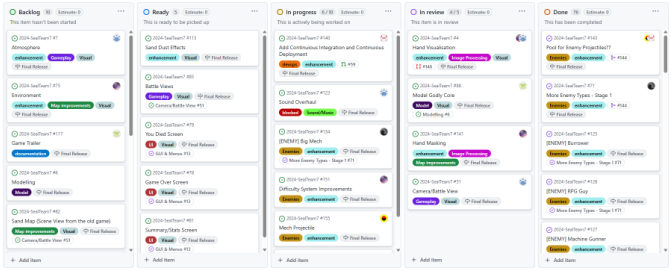


Fig. 5. Screenshot of Kanban board during project

6.7.2 Documentation

To maintain a consistent understanding across our project, a multi-faceted documentation approach was implemented. While code comments, Git commit messages, pull request descriptions, and the final technical report were crucial, our primary hub was a central, tabbed Google Docs document. This shared document acted as a knowledge base, with key information organised into sections such as:

- **Meeting Minutes:** Decisions, action items, and sprint discussions.
- **Task Brainstorming:** Early-stage feature breakdowns before formal GitHub issue creation.
- **Design Specifications:** Gameplay concepts, UI mockups, and design pivots.
- **Technical Research:** Notes on technologies like Kinect SDK, MediaPipe, IPC methods, and shaders.
- **Setup Guides:** Instructions for rig calibration, Kinect setup, and environment configuration.

Although we considered tools like Notion and GitHub Wikis, Google Docs's free access, real-time collaboration, and team familiarity made it the most practical choice. Maintaining it was a shared responsibility, with updates made live during meetings or shortly after major developments.

7 Individual Contributions

7.1 Peter Lillistone—Project Manager

Project Management & Leadership:

- Scrum master for most sprint meetings and retrospectives, leading proceedings and writing points on the whiteboard.
- Allocating issues based on people's preferences and expertise, also following up on tasks and making sure that team members are happy with what they are working on
- Liaising with academics/support staff to organise equipment, meetings or sorting logistics.
- Leading weekly feedback meetings with academics, helping relay project progress.
- Researched and ordered room decorations for games day and planned the room layout.
- Managing large-scale progress of the project, making sure that the team reached key milestones and that all aspects of development were on track.
- Checking in on individual progress, offering suggestions on how to resolve issues and a large amount of pair programming to help fix problems.

Hand Reconstruction

- Researched virtual hand reconstruction techniques used in other virtual environments.
- Found and rigged hand model using Blender armature.
- Implemented hand reconstruction code to calculate relative joint angles.
- Implemented landmark parsing code that predicted actual hand depth by cross-referencing the Kinect depth image and adjusting accordingly.
- Created custom shader and particle effect for hands.

Kinect

- Research and set up the Kinect to work with Unity, including testing available SDK and package support on different devices.
- Developed initial Kinect code to get asynchronous depth data into Unity.
- Performed frequent optimisations to improve Kinect performance, such as implementing shared memory models and improved parallelisation.
- In charge of calibrating Kinect when the setup was moved or relocated.

Music/Sound Effects

- Initialised Wwise into the Unity project.
- Added music and sound effects received from composers and found additional sound effects required for the game.
- Implemented all sound trigger code and volume control functionality.
- Sound balancing and testing of audio setup on rig.

Testing/Reviewing

- Ran numerous testing sessions, explaining and managing the setup and asking questions whilst individuals played.
- Analysed feedback from testing sessions and panels to help focus discussions in weekly sprints.
- Tested team members' code regularly and gave constructive feedback.
- Created bug reports on the Kanban board and distributed them to be fixed
- Compiling and fixing bugs for releases.
- Reviewed PRs and helped resolve merge conflicts.

Visuals/UI

- Implement a number of shaders, including boundary shader, hand shader and static shader.
- Created news feed UI, scrolling headlines and secondary display setup.
- Handled post-processing effects throughout the project, especially main game volume and newsfeed volume.
- Implemented dynamic camera creation and enemy tracking code for news feed UI.
- Created initial game UI and menus before redesign.
- Wrote menu and UI code that was later modified by Harry Greentree.

Sandbox Construction and Management

- Bought and installed the final box base with Bhagavath Achani.
- Researched and ordered the initial testing box.
- Contacted and organised acquiring sand from the concrete department, and experimented with different sand options for the final game.
- Led sand management process, monitoring the moisture levels of the sand for testing and preparing at the start of the day.
- Working with the support staff and team to get equipment for the setup and complete the required paperwork.

Map Generation

- Assisted Finn Cooper with optimising map generation code and performing profiling.
- Created terrain testing material to better display mesh heights.
- Worked on bug fixes in terrain mesh creation caused by LODs, resulting in broken chunks.
- Created wall models and placed them into the world, which were used up until background sand chunks were implemented.

Multiplayer (Scrapped)

- Researched different multiplayer alternatives and presented options for the group to decide on.
- Setup the initial multiplayer setup with the Kinect, alongside Finn Cooper, connected this to map generation.
- Wrote client and server connection code to allow for multiple players to connect.
- Took existing movement code and adapted it to work in multiplayer with multiple connected clients.
- Created basic multiplayer connecting UI to allow users to connect to specific servers.
- Experimented with temporal median filter as a noise reduction technique and established it was too computationally expensive for our use case.

7.2 Harry Greentree—Lead Programmer

Technical Leadership
- Established technical foundation by creating the GitHub repository and initialising the Unity project.
- Wrote unit tests to verify functionality and ensure new features didn't impact the existing codebase.
- Continually reviewed pull requests to maintain code integrity.
- Defined code style guidelines for consistency in the codebase.
- Implemented and enforced file structuring conventions.
- Set up CI/CD pipelines for automated testing and deployment, reducing errors and speeding up development.
- Enforced branch protection with PRs, no force pushes, and two reviewers to ensure code quality and safety.
- Created pull request and issue templates to ensure consistent and detailed information for the team to track.

Project Management & Leadership
- Facilitated communication with composers to align audio elements with gameplay.
- Conducted risk assessments to identify and mitigate potential issues
- Completed several requisition forms to acquire materials such as sand.
- Led some sprint meetings to ensure that development was on track.

Networked Weapon System (Scrapped)
- Implemented player controls for controller (Xbox) and mouse/keyboard using Unity's Input System.
- Designed a flexible weapon system using Scriptable Objects for easy creation and modification.
- Developed foundational base classes for weapon functionality.
- Implemented core weapon mechanics, including reloading, weapon swapping, ammo, and shooting.
- Added player feedback through sound and visual effects.
- Implemented hitscan logic for instant hit detection using Unity's raycast system.
- Researched FishNet (networking library) for multiplayer functionality.
- Utilised Remote Procedure Calls (RPCs) to synchronise weapon states between the server and clients.
- Implemented server-authoritative hit detection logic to ensure fairness and prevent cheating.
- Triggered particle effects and sound effects on clients for responsive feedback.

Core Gameplay Development
- Developed the core game manager to oversee the main gameplay loop.
- Implemented game state tracking.
- Integrated the core game timer.
- Implemented tracking for the shared player health pool.
- Added score calculation and tracking functionality.
- Added functionality to track defeated enemies.

User Testing Coordination
- Organised comprehensive play testing sessions with 20+ participants.
- Prepared and managed consent forms and participant information sheets.
- Systematically gathered and analysed player feedback and performance data.

Wave Difficulty System (Superseded)
- Created custom mathematical equations to dynamically scale wave difficulty.
- Tuned and adjusted the difficulty system to match the wave system.

Pooling
- Implemented enemy and projectile pooling systems to reuse inactive game objects instead of instantiating new ones during gameplay, improving performance.

Technical Exploration and Optimisation (Scrapped)
- Attempted to use Linux (Ubuntu 24.04) as the operating system for the game.
- Changed the Azure SDK library to use a C#-compatible wrapper.
- Wrote external function declarations in C# to interface with `libc` library.
- Fixed platform-specific rendering issues, including shadows on Linux.
- Identified performance bottlenecks in Unity on Linux with newer features.

Trailer & Academic Video
- Collected technical and trailer footage for use in the submitted video report.
- Narrated the entire technical video explaining the features of our game.

User Interface Implementation
- Collaborated closely with the lead designer to translate UI concepts into functional interfaces.
- Utilised assets provided by the lead designer to implement the complete custom UI design.
- Developed key UI screens, including the main menu, settings menu with tab navigation, in-game HUD, and game-over screen.
- Integrated UI elements with core game logic for seamless data display and interaction.
- Ensured a responsive and intuitive user experience across all UI components.

Leaderboard & Website
- Developed and implemented a cross-platform database solution using Firebase Realtime Database.
- Created a companion website using Flutter.
- Added full responsiveness across multiple device form factors.
- Implemented a dynamic leaderboard updating in real-time.
- Maintained consistent visual identity between the game and web platforms.

7.3 Finn Cooper

Active Team Member
- Frequently reviewed fellow team members' pull requests to ensure code coherence and functionality.
- Actively made use of the Kanban Board, creating new issues when when bugs arose or new features were needed as well as picking up tasks for myself.
- Play-tested the game often, providing feedback to team members and contributing to game polish.
- Pair-programmed with team members, to assist with integrating new features into the existing codebase.

Map Implementation
- Researched multiple methods for dynamic mesh generation in Unity to kickstart our key technology.
- Implemented a configurable mesh construction system using Burst compilation and the Unity job system.
- Created a noise generator class to enable testing when the Kinect is unavailable, improving team productivity.
- Worked with Sergi Lange-Soler in integrating 'background chunks' to extend the desert landscape outside the play area.

Map Optimisation
- Produced a Map Manager to govern map generation and ease development.
- Researched and implemented map chunking to allow for further performance improvements.
- Constructed a level-of-detail (LOD) system for chunk meshes.
- Developed a method of switching LODs based on the distance to the player (scrapped).
- Tested the performance of different map and chunk sizes.
- Pair-programmed with Sergi Lange-Soler to decouple mesh update from in-game framerate.
- Developed solutions to mitigate expensive collider mesh baking and tangent recalculation.

Multiplayer Systems (Scrapped)
- Worked with Peter Lillistone to set up and test basic client-server functionality.
- Adjusted map generation code to work over a server-authoritative network.
- Delved into FishNet documentation to troubleshoot various multiplayer issues.
- Made use of the FishNet Network Manager to synchronise the position and rotation of enemies.
- Used custom Remote Procedure Calls to spawn enemies and synchronise their state between the server and any connected clients.
- Utilised SyncVars to update and synchronise each enemy's attributes across clients.
- Implemented a rudimentary multiplayer-safe UI to display enemy health for each client.

Enemy Implementation
- Performed research into Finite State Machine (FSM) based enemies and designed basic enemy framework.
- Developed an abstract enemy class containing common enemy functionality, such as change in state, movement and on-death effects.
- Developed further enemy classes implementing the base class, each with custom attack effects, death triggers etc.
- Brought the first enemies to the game: the Soldier and the Tank, with distinct movement characteristics.
- Adjusted cargo plane and chinook enemies to fit the team's enemy design outline.
- Collaborated with Josh Chatten to refine enemy behaviours, including death conditions and movement force.

Enemy Spawning
- Developed configurable difficulty options, including scaling mechanics and configurable enemy spawn chances (superseded).
- Developed an Enemy Manager to handle the spawning and culling of enemies.
- Researched methods for pooling Unity GameObjects and produced template code for an enemy pool.
- Helped Sergi Lange-Soler to integrate his final difficulty system with existing EnemyManager code.

Enemy Pathfinding
- Explored and implemented A* implementations in C#.
- Created a custom node class and generated a representation of the map as a grid of nodes.
- Developed a `PathRequest` queue to allow for asynchronous pathfinding.
- Devised individual enemy heuristic functions to diversify gameplay.
- Visualised enemy paths in the scene view, to assist the team in debugging.

Toast Message System
- Developed a system for displaying messages in a UI pop-up during gameplay.
- Used this system to enable tips to appear upon seeing an enemy type for the first time.
- Helped Sergi Lange-Soler to integrate this system into a tutorial mode.

Visual Improvements
- Crafted several iterations of a 'toon' shader using Unity's shader graph to apply to all enemies and structures.
- Created a modified version of the toon shader with more colour bands to look good on our moving sand.
- Came up with a projectile system to synchronise visual feedback of enemy gunfire and damage taken.
- Designed projectile models using Unity primitives and particle effects.
- Implemented custom Mortar Tank projectiles, which use the Unity physics system to follow a planned trajectory.
- Used the Unity particle system to improve Tank and Burrower tracks.

Documentation
- Wrote a script for the terrain generation and enemy pathfinding sections of the academic video.
- Contributed to this report!

7.4 Bhagavath Achani—Lead Designer

Initial Mock Ups and Visualisations

- Curated extensive visual references, colour palettes, and hand-drawn concept art to establish artistic style and visual direction.
- Developed early mock-ups of player characters and gun designs to provide clear references for later development.
- Iterated through multiple logo designs to refine the game's brand identity.

Documentation and Communication

- Prepared and submitted the initial risk assessment documentation to secure workspace in the Big Lab, Queens Building.
- Identified potential sand-management hazards related to the project environment and equipment.
- Outlined appropriate safety precautions for both the lab space and the physical hardware used.
- Launched a Discord server to streamline team communication with organised channels for development, meetings and coordination.
- Initialised a shared Google Docs page for collaborative documentation of meetings, tasks, and research.

Box Construction

- Created technical drawings with accurate measurements and assembly specifications.
- Consulted with James Filbin (Hackspace Technician) to validate structural integrity and construction feasibility, given that the box housed 165kg of wet sand.
- Consulted with Amy Bland (General Engineering Lab Technician) to optimise material and parts selection as well as get construction insights.
- Sourced appropriate materials based on technical requirements and budget constraints.
- Managed the procurement process, including vendor selection and pickup after delivery.
- Completed an induction course to operate the laser cutter safely and independently.
- Coordinated with the Mechanical Engineering lab technician to cut plexiglass to precise dimensions.
- Troubleshot design challenges and implemented adjustments during and after the construction phase.
- Finalised the physical box construction with attention to aesthetics and user interaction

User Interface and Assets

- Hand-drew 25+ custom UI assets for the main menu page, settings pages and the game over page.
- Presented UI mockups during daily meetings to gather constructive criticism and improvement suggestions.
- Collaborated with the Lead Programmer to ensure UI assets functioned seamlessly within the game framework.
- Maintained consistent toon style across all UI components to reinforce the game's unique visual identity.
- Documented UI evolution through design versions to track improvements and reasoning.

Promotional Material

- Designed and produced comprehensive promotional materials through an iterative design process, incorporating weekly feedback cycles with team members.
- Developed poster designs for promoting our game and multiple sticker iterations featuring game logo and character artwork for distribution on Games Day.
- Designed posters showcasing enemy types with gameplay tips and visual identification guides.
- Designed custom t-shirts that incorporate game branding and signature visual elements.
- Integrated QR codes into poster designs and the game menu to drive traffic directly to the game website, enhancing digital engagement.

User Testing

- Created a comprehensive questionnaire with targeted questions addressing gameplay mechanics, visual elements, and overall user experience.
- Transcribed all user feedback, which was further referenced during weekly briefings and development.
- Organised structured testing sessions with Software Engineering Project Groups.
- Arranged evaluation meetings with panel members to receive more focused feedback.

Blender Modelling

- Created realistic gun models for the initial game idea.
- Researched VRC shader implementation options for creating realistic player models. However, it was discarded as we were using Universal Rendering Pipeline (URP).
- Evaluated alternative shader solutions that would work within our URP framework.
- Rigged and animated player models, ensuring they were game-ready with walk cycles, idle animations, and combat movements.
- Designed 20+ original colour-coded Blender models aligned with the updated visual aesthetic.
- Iterated on designs based on team feedback to perfect the toon aesthetic.

Academic Trailer

- Collected footage with fellow team members to highlight the game's features, atmosphere, and underlying technology.
- Edited the academic trailer to produce an engaging promotional and technical video
- Produced tailored trailers for musicians to feature in their personal portfolios.

3D Printing

- Experimented with physical 3D-printed models of the game core to create a frame of reference in the sandbox.
- Prepared digital models for printing using Ultimaker Cura slicing software.
- Troubleshot under-extrusion challenges by consulting with Max Tin (Hackspace Technician) for professional feedback to refine print quality and durability.

7.5 Sergi Lange-Soler

Masking

- Developed a novel method to identify the user's hands, arm and head in the depth image, using a combination of thresholding, dilation, vector arithmetic and ML-driven computer vision.
- Experimented with using the stability of pixels to determine whether they should be masked, the idea being that the user's hands, arms and head would be moving, whereas the sand would be mostly stationary.
- Implemented a method of masking out those regions from being reflected in the in-game terrain. This was done by keeping the height the same as it was on the previous frame.
- Experimented with various ways of avoiding temporary hiccups in MediaPipe's output from affecting the masking of the hands. Ideas included dropping frames, and calculating a projected set of landmarks from the previous two frames.

Noise Reduction

- Researched image processing techniques for noise reduction, such as the exponential moving average, Holt-linear and Holt-Winters smoothing, the Gaussian filter, the box/average filter, the median filter and low-pass filters in general.
- Tuned the linear interpolation/EMA parameter α, as well as the Gaussian blur radius σ, based on our team's aesthetic judgement and the user feedback. This had to be re-tuned when we switched to the play sand.
- Experimented with selecting α adaptively as a function of the difference in depth. First tried scaling it linearly, but later switched to a simpler threshold-based approach.
- Experimented with reducing the flickering effect using a similarity threshold, whereby the height is kept the same if it has only changed by a small amount.

Hand Landmarking

- Developed initial hand landmarking prototype with MediaPipe and OpenCV.
- Developed a system to handle converting the landmarks between 3 different coordinate systems: the downscaled, cropped image given to MediaPipe, the 1920×1080 image captured by the Kinect, and the 3D, 880×880, horizontally-flipped Unity world.
- Corrected the vertical position and scaling of the landmarks, by cross-referencing the wrist landmark with the depth image and scaling and offsetting the y coordinates by a certain factor.
- Developed debugging utilities such as gizmos which display the final hand landmarks and their connections right inside the Unity world, and a 2D visualisation window which shows the raw input and output for MediaPipe

C#-Python IPC

- Researched a wide variety of IPC methods, including data transfer methods such as TCP sockets, shared memory segments, memory-mapped files and named pipes; synchronisation methods such as socket-based control signals, Win32 Events and POSIX semaphores; and IPC libraries such as ZeroMQ and gRPC.
- Developed an initial implementation using sockets, JSON serialisation and deserialisation and a client-server model.
- Profiled the code and identified that the IPC was a key bottleneck in the terrain mapping pipeline.
- Rewrote the Python script and the `PythonManager` class to transfer data using shared memory (implemented using memory-mapped files), and synchronise access using Win32 Events.
- Replaced expensive method invocations with manual pointer arithmetic to reduce data copying as much as possible.
- Added Linux compatibility, using polymorphism and an abstract base class to encapsulate the platform-dependent APIs. Data transfer on Linux was also implemented using memory-mapped files, whereas synchronisation was achieved with POSIX semaphores.

Background Terrain Interpolation

- Replaced our previous solid blue background with procedurally-generated "background chunks" which extend from the depth-mapped "play region" to the edge of the screen.
- Implemented two components: (1) a large-scale sample of Perlin noise, for the hills and valleys; (2) a small-scale sample, for the finer details.
- Researched interpolation methods such as smoothstep and Hermite interpolation. Initially, it was necessary to use the general form of cubic Hermite interpolation, as I had to match the gradient on the end-points. Later, I switched to the zero gradient, "smoothstep" form due to large gradients in the play region.
- Developed a system to interpolate between the play region chunks and the background chunks along all 4 sides. This involved some fairly complex calculations due to the variable levels of detail and the difficulty of getting a smooth result in the corners.

Final wave-based spawning system

- Identified a gameplay issue with our game—the fact that enemies spawned randomly and appeared to be completely mixed rather than remaining in groups.
- Developed a custom C# DSL for programmatically specifying the design of each wave.
- Created abstractions for groups of enemies in arrangements such as an $n \times m$ grid (used for the human enemies) and a row of enemies spawning at regular intervals (used for the vehicles).
- Used the DSL to design the patterns of enemies to spawn over the course of the game, with the intention of creating an impression of a coordinated attack.
- Designed the tutorial and several toast messages to teach the user how to play and address common points of confusion.

7.6 Josh Chatten

Enemy Physics

- Created and tested many iterations of a physics system in which enemies interact with the Unity Rigidbidy physics as well as custom functions to aide with the shortcomings that the Unity physics engine has with dynamic/deforming meshes.
- Creation of an abstract class for scripting physics and then inheriting multiple different abstract children to differentiate between ground units, aerial units and underground units. Each specific enemy type then inherits these to make smaller changes.
- implementation of different death mechanics and attack methods for enemies
- Improved the feeling of interacting with enemies by introducing the ability to send enemies flying up. This was done using a Laplace distribution to determine both whether an enemy is sent flying and then using the distribution graph itself generate an angle that enemy is pushed.
- Created an explosion simulation to push enemies away from vehicles that explode. It was tuned down quite substantially in the final game, but was still present.
- Lots of testing was employed especially when it came to the vehicles to make sure they can move freely and correctly. This was done in tandem with Finn.

Concept, Creation and Implementation of Enemy Types

- Creation of over 20 prototype enemy types, each with their own mostly unique gameplay.
- Testing and Filtering of enemies to only include the polished and fun ones in the final product
- Implementation of enemy mechanics including: burrowing, exploding, enemy summoning, resurrection, etc...
- Explored and then fully implemented a colour scheme to separate enemies based on related functions and looks.
- Created base models and concepts for enemies to then be expanded on, and detailed models created.

Visual Effects and Particles

- Creation of initial particle effects for enemy shooting. These were later changed to be real projectiles instead
- Creation of interesting and unique visual effects using VFX graph, such as explosions and the glowing core on the tower.
- Creation of some other particle effects, such as dust trail for tanks and burrowers
- Small adaptations were made to enemy physics and AI to allow better viewing of these visuals. Especially after death(s).

Research and Theory Crafting

- Research into possible interactions with the sand.
- Theory crafting game modes and gameplay (both group and solo).
- Research the best ways to create visual effects.
- Theory crafting enhancements to different aspects of the game based on feedback (such as the Laplace-propulsion and colour coordination of enemies)

Design, Concept and Miscellaneous tasks

- Drew the very first logo for SealTeam7.
- Hand-drew concept art for both original game pitches using Krita (A free open source art program).
- Organised the initial lore and naming of the game (pre and post pivot).
- Recorded 10 minutes of in-game footage using custom camera placement for the trailer portion of the video.
- Assisted in scripting and organising the layout of the academic video.
- Often took notes during sprint meetings and took charge during conceptual parts of the meetings, especially when talking about physics, enemies, or general gameplay elements.
- Pair-programmed with team members to help fix difficult bugs and align our understanding of the direction we wanted the game to go.
- Led many internal testing sessions focused around features and enemy physics.
- Undertook many tests to find bugs and then subsequently report them, fix them, or both.

Created and Implemented Final Boss (Scrapped)

- Created a basic blocky model of a mech.
- Implemented feature where the mech shoots and turns into a ball.
- Implemented a physics playground experience where to kill the mech, you must roll the ball using the sand into its own missile.
- Researched, tested, and then created an entirely new physics script for the ball to allow it to move along the sand like it is being pushed and roll along rough, uneven and dynamic terrain.
- Partially tested, but also kept secret as a hopeful surprise for the panel. However, we as a team agreed that it was untested and a model lacking in detail; as such, it would be better to leave out the final game delivery.

Designed and Implemented a Full FPS Movement System (Scrapped)

- Researched modern FPS games and took notes on what movement felt good and what to avoid.
- Designed a fast, paced and fluid series of movement mechanics for a first-person shooter game (sprinting, sliding, wall running, etc).
- Implemented this system using Rigidbody Physics.
- Tested and tuned this system for both Xbox controllers and mouse and keyboard. (With help from Harry).
- Added features that improved the feeling of going up and down slopes, and was partway through testing both multiplayer and the movement when paired with the sandbox.
- Was partway through creating a HUD and other UI for this first-person shooter.
- This was scrapped during the pivot, but is still very polished and fun, and can very easily be taken and used in other projects in the future.

8 Software, Tools & Development

8.1 Git & GitHub

8.1.1 Branches & Pull Requests

Our development workflow relied heavily on Git branches and GitHub Pull Requests (PRs) to manage code changes and maintain stability. We established a core branching model consisting of a `main` branch, representing the current stable build of the game, and a `dev` branch, serving as the primary integration branch for ongoing development. Branch protection was enforced on these critical branches, requiring that all changes be made through a PR to safeguard against broken code being integrated into the stable codebase.

New features, bug fixes or experimental work were developed in dedicated feature branches, which were created directly from the `dev` branch. We often initiated draft PRs early to track features in progress and facilitate preliminary discussions. Once development of a feature was complete, the draft PR was converted to a formal PR targeting the `dev` branch. Merging a PR required a vigorous review process, requiring at least two reviews from team members. This review process frequently involved reviewers checking out the feature branch locally, building the project, and performing hands-on, often in-person, testing to ensure functionality and code quality before integration (See Section 8.9.1).

8.1.2 Continuous Integration & Deployment

GitHub Actions were used for our CI/CD pipeline to automate linting, unit testing, and build verification. We integrated the GameCI framework [2] to handle Unity builds within GitHub Actions workflows. The pipeline was configured to target Windows and Linux platforms exclusively, aligning with our deployment goals and the native support of our core dependencies. Unit tests were written to validate key gameplay and system functionality, helping to ensure correctness throughout development and catch regressions early. This setup provided fast feedback on pull requests and consistent build validation without manual intervention, streamlining team collaboration.

8.1.3 Git Large File Storage

Git Large File Storage (LFS) was a crucial development tool for managing our game's large binary assets. We used Git LFS because several binary files exceeded GitHub's size limitations for standard repositories. By configuring our `.gitattributes` file to track bundle files and compiled native libraries (`.so` files), we were able to maintain version control efficiency while ensuring a consistent development environment across our team. This solution allowed us to track changes to large Unity-generated assets without bloating the repository or encountering push failures, streamlining our collaborative workflow throughout the development process.

8.2 Unity

Unity is a widely used game engine, supporting cross-platform development with an array of tools and extensive documentation. It was chosen for this project due to its lightweight design and high level of customisability, which is not as present in other similar engines. Unity also has good support for mixed tech stacks and handling external peripherals, making it most suitable for the project.

8.2.1 Unity Debugger & Profiler

Unity has a suite of built-in profiling and debugging tools which were invaluable for carrying out performance optimisations throughout the project [3]. The built-in profiler includes features such as memory management breakdown, core utilisation, and, most usefully, deep profiling, which allows for compute times for individual lines. These features have been especially useful in iteratively optimising Kinect and map generation code, which is crucial to maintaining responsiveness in our game.

Fig. 6. Example Unity profiler readout from project

8.2.2 Package Management

NuGet for Unity was employed to integrate third-party C# libraries and wrappers critical to the project's functionality, including EmguCV (a .NET wrapper for OpenCV), K4AdotNet (Azure Kinect SDK bindings), and Firebase with its Realtime Database module. NuGet's version control and dependency resolution streamlined library integration, ensuring consistency across the team's development environments.

Cross-platform compatibility was prioritised as the team had a wide variety of development environments and platforms. Most packages used supported both Windows and Linux natively. However, limitations arose with EmguCV, which lacked a freely available macOS (ARM64) native wrapper, necessitating a licensed solution for full compatibility. This constraint highlighted the importance of evaluating platform-specific dependencies early in the development lifecycle.

8.3 Python

Python was used to handle real-time hand-tracking, due to our team's prior experience with the MediaPipe framework and the convenience of its official Python API wrapper.

A separate Python process executed the MediaPipe model inference. Furthermore, the process orchestrated the shared memory interface, allowing for efficient and synchronised communication of inference results, such as hand landmarks, between itself and the Unity game engine.

8.4 Blender

Blender served as the primary 3D modelling software throughout the project, enabling the creation of all custom 3D assets. Initially, it was used to model realistic assets like guns and rigged player characters for the early FPS concept, though these were later scrapped following the project's pivot. Blender was instrumental in designing and producing over 20 distinct, colour-coded 3D models for the enemy types and the central core structure for the final game. Furthermore, Blender was also used to prepare digital models for 3D printing experiments, enabling the creation of physical objects intended for interactive use with the sandbox.

8.5 Procreate

Procreate was utilised extensively for creating the 2D visual elements that defined the game's look and feel, complementing the 3D assets. Its initial use involved generating hand-drawn concept art and visualisations to establish the overall artistic direction. Over 25 custom UI assets, such as components for the main menu, settings pages, and game over screen, were illustrated in Procreate. Additionally, Procreate facilitated the design of various promotional materials, such as t-shirt graphics, posters detailing enemy types, and stickers featuring the game's logo and characters.

8.6 Wwise

WWise is a Unity-integrated audio authoring software that implements robust game audio. It offers a wide range of features such as balancing suits, spatial audio capabilities, and a more in-depth wrapper than Unity's built-in system. It was used in the project to implement and balance all the sound effects and music.

8.7 Physical Box

Extensive user testing of our original sandbox, a 600mm × 800mm plastic container, revealed several significant shortcomings. From a practical standpoint, its height made reaching inside awkward, whilst its rectangular shape led to misalignment with the square digital map. Furthermore, users expressed a lack of immersion due to both the sandbox's visual appearance and the insufficient volume of sand it contained. These combined factors prompted us to construct a new, larger sandbox with improved functionality and aesthetic design.

To tackle this redesign, we consulted multiple academics across the Computer Science and Engineering departments.

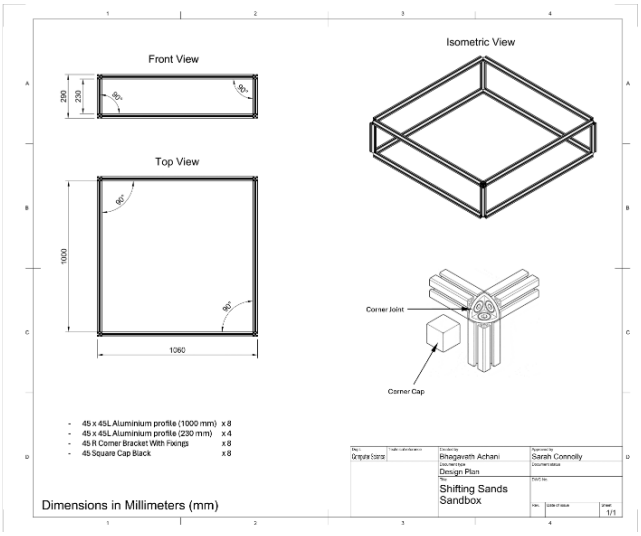


Fig. 7. Design Plan of Sandbox Skeleton

Our first discussion was with Dr. Chris Snider, an Associate Professor in Engineering Design, who provided initial guidance on constructing a box suitable for AR/VR integration. We also sought advice from James Filbin, the Hackspace Technical Manager, and Amy Bland, a specialist technician in the General Engineering Lab. They provided valuable insights regarding structural integrity, effective construction techniques and appropriate material selection to build a sturdy and functional box.

Given that the box needed to support up to 200kg of sand, structural integrity was a key concern. We chose 30mm wide aluminium extrusions for the frame to address this, as they provided the strength and rigidity required for such a load. We initially considered using wood for the side panels, but ultimately decided on plexiglass for a more polished appearance. Finally, we decided on dimensions of 1000mm × 1000mm for the box, as this square format would best suit our needs and matched the specifications used in the research paper we were referencing [1]. Following these decisions, we created detailed technical drawings using the software tool Autodesk Fusion (Figure 7), ensuring precise dimensions and design specifications. Once finalised, we selected vendors and sourced the necessary materials for construction.

8.7.1 Future Improvements

1) **Material Selection for Base:** Although the walls of the sandbox were constructed from plexiglass, the base was made from Medium-Density Fibreboard (MDF). However, when wet sand was introduced, the MDF base began to deform and deteriorate. Therefore, using more durable materials like treated wood or plastic could mitigate this issue in future iterations.
2) **Accessibility and Box Size:** While the increased size of the box improved gameplay, it also introduced accessibility challenges for players with shorter arms, who

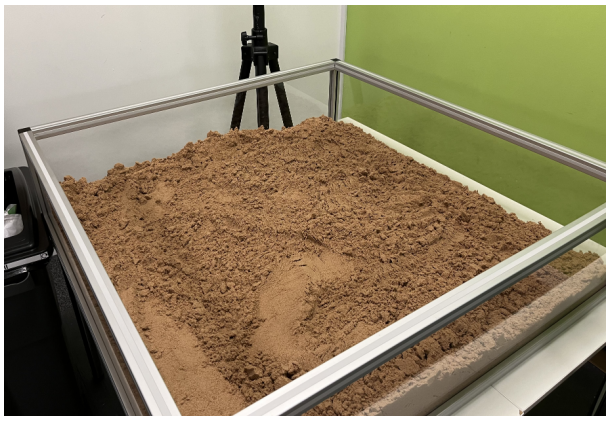


Fig. 8. Completed aluminium/plexiglass box with initial 90kg of play sand

found it difficult to reach the edges. In future designs, a balance between the box's width and player reachability should be explored, whilst ensuring the square dimensions are maintained for gameplay purposes.

3) **Sealing Method for Sand Leakage:** During initial playtesting, it was observed that sand leaked from the edges of the sandbox, an issue that worsened as the sand dried. Although attempts were made to seal the edges with a hot glue gun, this solution proved ineffective. A more reliable sealing method, such as silicone, should be considered for future implementations to prevent sand leakage.

8.8 Development

8.8.1 Code Standards

To ensure a high quality of code throughout the project, formal coding standards were implemented into the team's development workflow. Use of IDE linting, like that provided by Rider and our GitHub CI, was used to ensure consistent syntax and spacing throughout the code base. Thorough code reviews, as mentioned in the next section, also assisted in maintaining this standard.

8.8.2 Code Review

Along with code standards, a rigorous reviewing process was implemented requiring testing passing on the rig and 2 separate reviews giving approval on the quality of the code. This often resulted in multiple iterations of feedback and improvement on a feature before being accepted into the codebase, allowing us to ensure the stable state of the game throughout development.

8.8.3 Pair Programming

As mentioned in Section 6.2, the team endeavoured to be in the lab testing and developing five days a week. Because of this, we were able to conduct frequent pair programming sessions. This led to quicker development and faster problem resolution, significantly reducing the time spent debugging and implementing complex systems. Pair programming also proved invaluable when members worked on parts of the game they were less familiar with, as they could consult the expert on that part of the game.

8.9 Testing

Testing was an integral and continuous process throughout our development lifecycle, ensuring iterative improvements across several core aspects of the game. The primary objective was to gather extensive user feedback, focusing specifically on user interaction with the physical setup, intuitiveness of core gameplay mechanics, perceived level of challenge, and the overall player experience. This commitment to user-centred design is reflected in the scope of our testing, which included over 60 distinct testers and around 10 feedback sessions by the end of the development cycle. A dual testing approach of internal team testing and formal user testing was employed to support this process. These sessions yielded invaluable qualitative insights, allowing us to identify key user patterns and validate our design choices.

8.9.1 Internal Team Testing

The team committed to a rigorous internal testing schedule throughout the development cycle. At least once per week, a stable build of the game was compiled, reviewed and tested by the development team. This regular internal feedback loop was crucial for identifying and resolving bugs/issues early and enabling iterative refinement of features.

8.9.2 Formal User Testing

Formal user testing followed a structured, ethical protocol to ensure reliable and meaningful feedback. Each participant was first provided with a participant information sheet and a consent form, both of which were explained in detail before testing. Once consent was obtained, participants were asked to play the game for approximately 5 minutes, guided by a team member. Audio and video recordings were made throughout the session to capture player interactions, reactions and verbal feedback, which could then be used for post-session analysis.

After gameplay, participants were interviewed using a structured set of questions covering the physical interaction and the digital gameplay. This process provided a mix of direct observation and self-reported insights, enabling a comprehensive evaluation of the game's strengths and areas for improvement.

8.9.3 Results & Insights

Feedback from testing revealed several key insights that guided subsequent development. Testers consistently highlighted the novelty and immersiveness of shaping the terrain manually, with many describing the sandbox as "intuitive" and "fun to manipulate." This validated our core design decision to centre gameplay around tangible interaction. However, the testing process also uncovered several usability and gameplay issues.

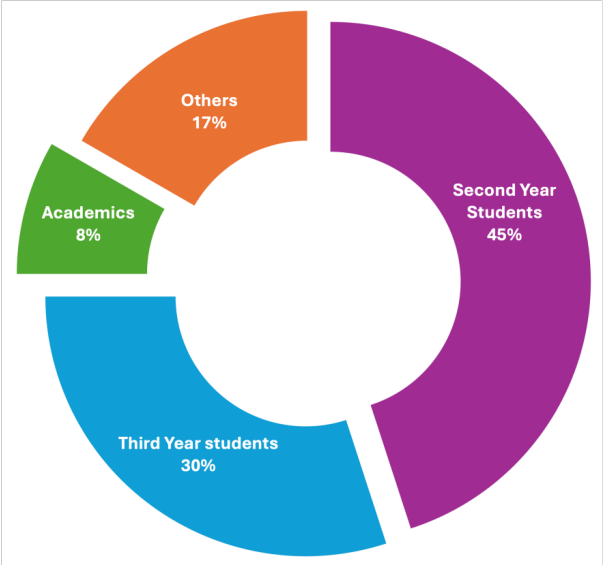


Fig. 9. Distribution of Users Across Testing Cycles

Tutorial Mode and Contextual Pop-Ups: From a gameplay perspective, our testing revealed opportunities for improved clarity and player guidance. Many participants expressed confusion about the game's objectives and enemy mechanics, with several unable to understand how to effectively combat different enemy types. In response, a tutorial mode was implemented, which gradually introduces enemy types, allowing the players to learn enemy mechanics. Additionally, informational pop-ups were added to suggest effective strategies for defeating specific enemies, providing players with guidance during gameplay.

Wave/Difficulty System: The difficulty progression received mixed feedback, with some describing it as "appropriately challenging" while others found the difficulty curve too steep, describing it as "exponential" rather than gradual. This issue was fixed by implementing a wave system, creating better pacing, as well as allowing players to strategise and modify their defences. Additionally, multiple testers suggested including visual indicators of progress such as wave counters and score displays to enhance engagement and provide clearer feedback.

Spatial Feedback: In terms of the physical setup, several testers expressed a desire for more explicit spatial cues to help them gauge where they were manipulating the sand. To explore this, we experimented with placing 3D-printed landmarks in the sandbox to serve as reference points. However, follow-up user testing revealed that these physical additions became obtrusive during play, ultimately leading us to discard the idea in favour of maintaining a clean and unobstructed environment. Instead, 3D models of hands were implemented, which track the player´s movements in real time, providing a non-intrusive spatial feedback within the sandbox.

9 Technical Content

9.1 Kinect Introduction

The Azure Kinect DK was Microsoft's all-in-one sensor solution aimed at offering a more professional option than previous iterations of Kinect devices. This project will mainly be making use of the 1MP Time-of-flight depth camera (used for accurately measuring the sand's height) and the 12MP CMOS RGB camera (used for hand landmarking and masking) [4], [5]. This device was mainly chosen because it was most similar to the original source papers' use of the Kinect v2 [1], whilst offering higher spec sensors and a more up-to-date SDK. It was also partially due to its availability in the department without having to order additional equipment, making it a fast option to start with.

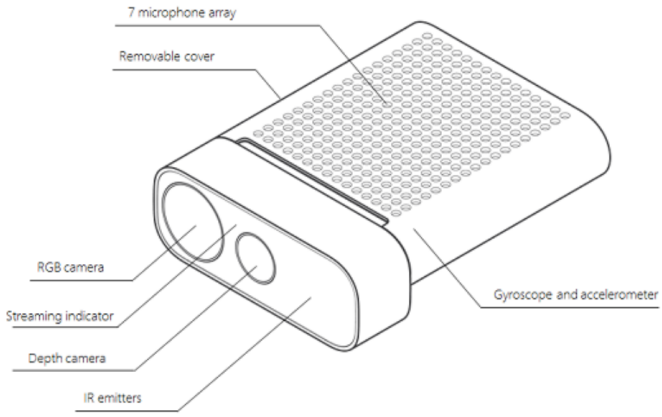


Fig. 10. Diagram of Azure Kinect DK sensor arrangements.

In retrospect, it might not have been the most optimal device for our implementation, mainly due to the following limitations with the technology:

1) Maximum Frame rate—the Kinect DK is limited to 30fps for all sensors, which means that the maximum achievable "true frame rate" in the game is 30fps. Whilst this can be mitigated by frame interpolation and asynchronous threading, it still limits the real-time nature of the game.
2) Noise in the depth sensor - the depth image has a "typical systematic error" of no more than 11mm $\pm$ 0.1% [5]. In practice this can result in fluctuations of 2–3mm between individual frames. Due to this, a number of image processing techniques have to be used to reduce this noise, which often increases latency.
3) Limited support and documentation - due to the Kinect DK being discontinued in 2023 and its limited shelf life, it has very little formal documentation of its API. This has led to a lot of reverse engineering from sparse code examples and third-party implementations.

Considering these limitations, future implementations might consider using higher-performance all-in-one sensors or separate higher-quality sensors. This would likely be preferable as individual sensors would likely perform better

and be cheaper, as they don't contain a load of unused sensors and features. However, this would require more complex calibration code, which in our case is mainly handled by the Kinect, but it would likely allow for better results with respect to all the limitations listed above.

Having chosen a depth sensing device, we need a method of bringing that data to our digital world. Figure 11 presents an overview of our solution, which future sections explore and explain further.

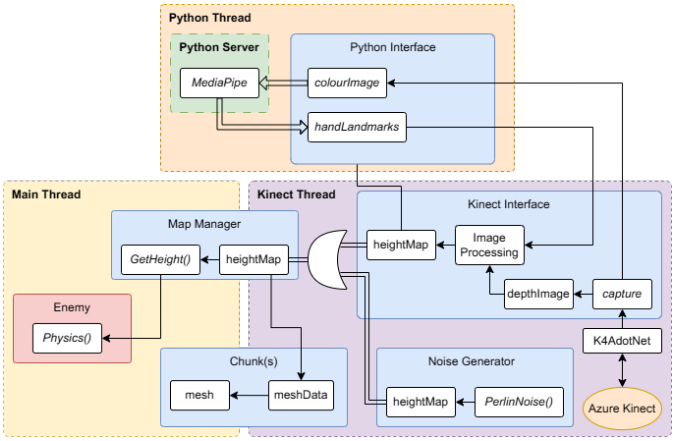


Fig. 11. Diagram describing overall system structure.

9.2 Kinect Interface

With the Kinect hooked up, we need a method of capturing its data. This is implemented as a Kinect Interface class which uses K4AdotNet, a C# wrapper for the Kinect API, to start, stop, and read from its sensors. The API's `Device.GetCapture()` method must be called every time we need a new image from the Kinect. This returns a capture containing a colour image and a depth image [6]. We therefore use this function Unity's `Update()` method, which is called every frame.

The first issue found is that if no new images are available, `GetCapture()` blocks the current thread until the Kinect captures a new image [6]. If called from the main Unity thread, this freezes the whole game, causing impractical stuttering. To rectify this, we move the `GetCapture()` calls to a loop on a separate background thread, using C#'s built-in `System.Threading` library.

Once a depth image is received, it must be processed as described in Section 9.3. We perform this processing on the same background thread as to avoid slowing down the main thread. Finally, the processed height map is written to a local array for use in terrain generation (Section 9.4).

9.2.1 Noise Generator

To facilitate rudimentary feature testing when the Kinect is unavailable, we make use of a `NoiseGenerator` to stand in place of the Kinect interface. This generates a grid of heights sampled from Unity's built-in `PerlinNoise` function, which progresses through time every frame, providing a moving surface somewhat analogous to real Kinect depths.

9.3 Image Processing & Masking

The raw depth image obtained from the Kinect must be processed before we can use it to update the in-game terrain. We discuss two key challenges: (1) the masking out of the user's hands, arms and head; and (2) the reduction of noise in the depth data.

Figure 12 shows our image processing pipeline. At a high level, we first use thresholding and MediaPipe's hand-landmarking solution to identify regions in the depth image that should be masked. Then, we use the exponential moving average (EMA; discussed in Section 9.3.2) and a low-pass filter to reduce noise, only updating the regions which are not in the mask.

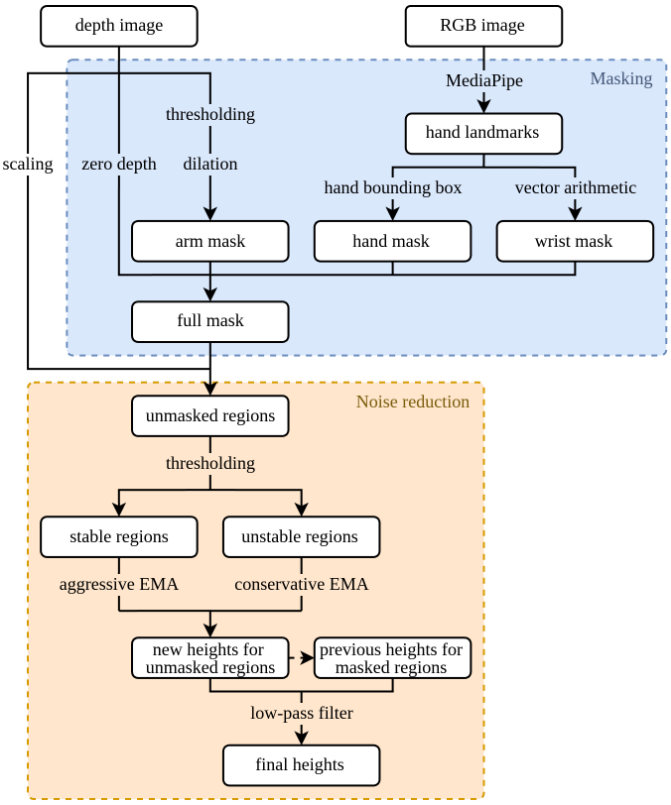


Fig. 12. Image processing pipeline. Arrows indicate processing steps, and boxes indicate images and other objects.

The dotted arrow above the low-pass filter step indicates that the previous heights for the masked regions are stored in the same image as the new heights for the unmasked regions. In other words, there is one image stored in memory that EMA writes to and the low-pass filter reads from; the new data is written to the unmasked regions, while the masked regions are kept as they were on the previous frame.

9.3.1 Masking Non-Sand Regions

As the user's hands, arms and head are often between the sand and the Kinect, it is necessary to somehow prevent these regions from affecting the topography of the in-game sand. This problem can be broken down into two parts: (1) identify those regions, and (2) fill them with alternative height values.

Our approach for identifying the regions to mask is as follows. We find low-depth (i.e., high-elevation) regions by thresholding the depth image. The result is dilated to avoid artefacts at the edges. We refer to this binary mask as the *arm mask*, though it is also effective for the head and the hands when they are below the threshold.

However, when the hands are near the sand, they fall above the threshold. To deal with this case, we use Google's MediaPipe library [7] together with the RGB image obtained from the Kinect to find the position of 21 key points (referred to as landmarks) on each of the user's hands. The bounding box of the landmarks for each hand constitutes the *hand mask*. (We also add padding to account for random noise in the landmarks.)

This leaves a problematic region around the wrist, which is often above the depth threshold but outside the bounding box of the landmarks. To mitigate this, we take the vector from landmark 9 to landmark 0 and add a scalar multiple to landmark 0 to estimate the location of this region. This vector arithmetic is illustrated in Figure 13. We add a square of a fixed size, centred on this location, to the mask. An alternative method would be to simply add a fixed vector to landmark 0, however this would not adapt to the rotation of the hand.

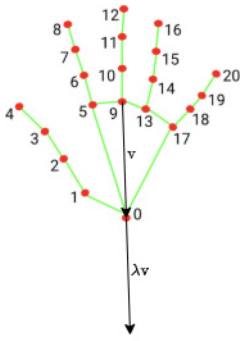


Fig. 13. The vector-arithmetic heuristic we use to estimate the location of the wrist. The vector from landmark 9 to landmark 0 is denoted v, so its scalar multiple is λv, where λ is a scalar constant. We add λv to the position of landmark 0 to find the wrist location, which is the tip of the lower arrow.

The Kinect is sometimes unable to get depth readings for certain pixels. This can occur when there is a sharp change in depth, such as on the boundary of the user's hand, or when an object is too close to the sensor. For these pixels, the value is set to zero in the depth image. To deal with this, we add them to the mask and use the same masking technique.

A diagram of the masks we use is shown in Figure 14. The arm mask captures the arms, with the extra padding resulting from the dilation; the hand mask is the bounding box of the hand landmarks obtained using MediaPipe; the wrist mask is calculated using certain vectors in the hand landmarks; and the zero mask consists of all pixels for which the Kinect could not find a depth. In this example, the Kinect could not find the depth for all pixels in the hand, due to it being outside of the Kinect's operating range.

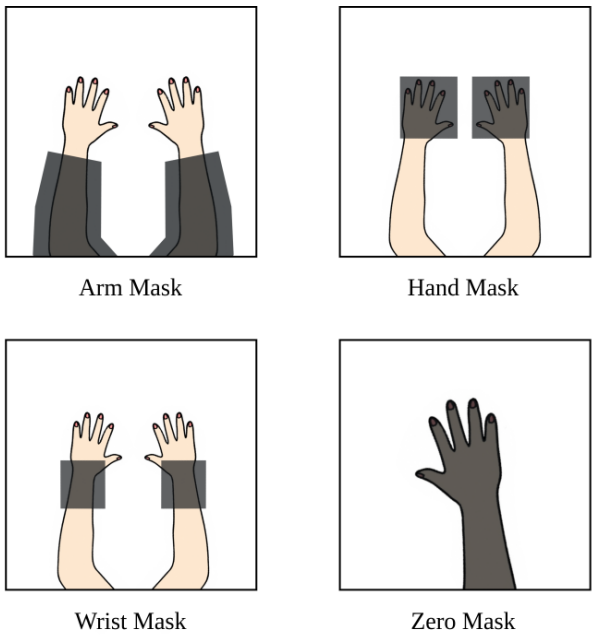


Fig. 14. Top-down view of the four masks we use to remove the user's hands, arms and head from the in-game sand.

This describes how we identify the regions to mask. To fill in the masked regions, our approach is to use the height value for each pixel at the last time it was not in the mask. As the height is left unchanged, we can implement this efficiently by writing the result of the exponential moving average (discussed below) into the same image we wrote into in the previous frame, skipping over all pixels in the mask.

This approach has some limitations. For example, under certain conditions, a wall-like artefact could be created in the in-game terrain, despite no such wall in the real-world sand. The conditions for this artefact to be created are as follows:

1) An object, such as a part of the user's arm, is positioned slightly below the arm-masking threshold, and is not masked out by any other method. This results in the object being captured in some region of the in-game terrain.
2) The same object, or a different object, moves above the arm-masking threshold, in a similar region of the depth image. It is not necessary for the regions to be precisely

the same; only for there to be some overlap. This results in the overlapped region being masked.

3) Due to our masking method described above, the heights in the overlapped region are kept the same as they were in the previous frame. Therefore, each pixel which was (erroneously) not masked will be frozen in place for as long as there is an object above the arm-masking threshold.

This artefact typically comes about when the user's hand (or the end of their arm) is not masked, and the user proceeds to move their arm forward, in such a way that their forearm moves towards the previous position of their hand. This process can repeat for as long as the user is moving their arm in this way, causing a wall-like artefact along the path taken by their hand.

Figure 15 explains this with the help of a diagram. In the first box, the user's hand is below the threshold, and we assume that MediaPipe was not successful in finding it. Therefore, the structure of the hand is recreated in the in-game terrain. In the second box, the user moves their arm forward, such that their arm is above the threshold in the region where their hand was previously. This causes the structure of the hand to be fixed in-place in the in-game terrain, as it was on the previous frame. This can continue in the same way if the hand remains at a similar height and the whole arm continues moving in the same direction.

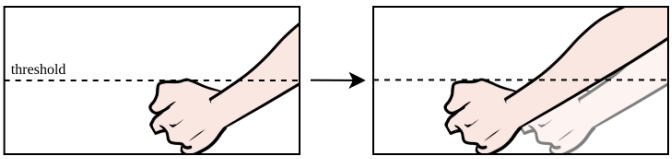


Fig. 15. Side-on view of the "wall" artifact.

9.3.2 Noise Reduction

The second key challenge with the raw data from the Kinect is that, as it uses a physical sensor, there is substantial random noise. As discussed in Section 9.1, the noise has a standard deviation of up to 17 mm. If we recreate the raw data in Unity, the terrain is far more bumpy than the ground-truth, with the bumps changing rapidly from one frame to the next.

We refer to this effect, where the recreated terrain visibly changes in regions where the sand hasn't been moved, as *flickering*. Without any processing to reduce it, flickering looks like many small bumps, which are randomised every frame. If it is mostly eliminated, the terrain appears relatively smooth and constant, but the effect is still perceptible as random differences in lighting between frames. After a long process of iteration and experimentation, we reduced the flickering effect to the point of not being visible to the user.

To reduce flickering, we first use an image processing technique known as *exponential smoothing* or the *exponential moving average* (EMA). [8] EMA is defined by the equation

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1} \tag{1}$$

where s_t and x_t refer to the smoothed and observed values at time t, and α is a constant known as the *smoothing factor*. It can be seen as a weighted average of the latest observation and the previous smoothed value.

Alternatively, by recursive substitution, it can be shown that

$$s_t = \alpha x_t + \alpha(1-\alpha)x_{t-1} + (1-\alpha)^2 s_{t-2} \tag{2}$$

$$= \alpha[x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + \cdots]. \tag{3}$$

In other words, EMA calculates the weighted average of all the observations, with the weights decreasing exponentially according to time.

Rather than implementing Equation 1 manually, we use Unity's `Mathf.Lerp` method [9] as it is optimised for performance. We use it to linearly interpolate between the value previously stored (s_{t-1}) and the new observation (x_t) with the interpolation parameter t set to our chosen value for α. This is equivalent to EMA because the equation for linear interpolation,

$$s_t = s_{t-1} + \alpha(x_t - s_{t-1}), \tag{4}$$

can be rearranged to make Equation 1.

We find EMA to be very effective at reducing flickering; however, it has a detrimental effect on responsiveness. The technique is known to create a "lagging" effect, due to the fact that the previous observations form part of the average. Worse, rapid changes in the value sometimes disappear completely, as the moving average does not have time to catch up.

In our user testing and the feedback from the MVP and beta panels, it was noted that good responsiveness was key for a good user experience. When responsiveness was low, users would sometimes move a heap of sand too quickly for it to change the in-game terrain significantly, which appeared to lead to confusion and frustration. We worked around this temporarily by advising testers that it helps to make slow, deliberate movements. However, by games day, this strategy was no longer necessary, and in fact, we saw panel members successfully making rapid movements such as passing a heap of sand between their hands.

In our testing, as we tuned the value for α, there appeared to be a trade-off between low flickering and high responsiveness. It was not possible to eliminate the flickering entirely while retaining good responsiveness just by selecting the right α.

To avoid this trade-off, our technique is to vary the value of α based on the stability of each pixel. If, for a given pixel, the observed value has changed by a large amount, we prioritise responsiveness and choose a large α. We refer to this as

conservative or *attenuated* EMA. This causes the smoothed result to be more heavily weighted towards the most recent observation, thereby increasing responsiveness at the cost of increased flickering. If, on the other hand, the observed value has only changed slightly, we prioritise low flickering and choose a small α (*aggressive* EMA). The result is that, in the regions where the sand is being moved (which we refer to as *unstable* regions), the changes in the topography are quickly realised in the in-game terrain. Meanwhile, in the regions where the sand is stationary, the noise is smoothed out to the point of being imperceptible. To our knowledge, this technique has not been discussed in the image processing literature.

By applying EMA, we are essentially exploiting nearby data in the *temporal* dimension to smooth or "average out" the noise. It is the same principle as used in scientific measurement—we take the average of multiple observations, reducing random variations due to imprecise equipment. The other noise reduction technique we use, low-pass filtering, exploits nearby data in the *spatial* dimension to do the same thing.

There are many different low-pass filters identified in the literature. We chose the Gaussian filter because we believe it is better suited to our data. For instance, one alternative, the median filter, is typically used to reduce "salt and pepper noise"—random black and white pixels in the image. Taking the median can eliminate outlier values completely, even with a small kernel size. On the other hand, a filter that finds a weighted average, such as the Gaussian filter or the box filter, will always be influenced by the outlier pixel. However, for our application, it is evident from our testing that there are no outliers present in the image at the end of our masking process. Therefore, for the same reason that the mean is a better statistical summary than the median when the data are not skewed and there are no outliers, the Gaussian filter is better suited to our needs than the median filter.

As shown in Figure 12, we apply the Gaussian filter to the whole image, not just the unmasked regions. This is in part for convenience, as OpenCV [10] does not have the ability to apply a Gaussian filter only to certain pixels in an image, but also to smooth over a staircase effect caused by the masking.

Figure 16 illustrates the effect of our noise reduction technique on improving the quality of the in-game terrain. As visible from the picture, our two-pronged approach drastically reduces the noise present in the raw depth data. It is not clear from this static image, but if one observes the side-by-side comparison in the video report, one can see that we also reduce the flickering effect to a negligible amount. If the Kinect has any built-in noise reduction, ours appears to be more effective, without causing lag or other negative effects.

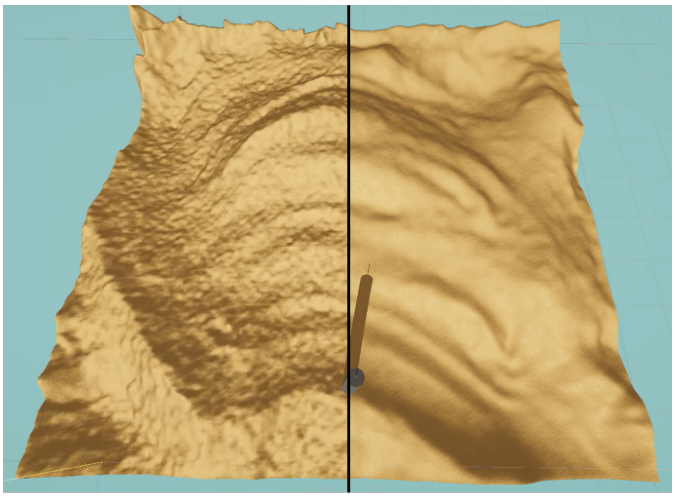


Fig. 16. Side-by-side comparison of noisy height data and de-noised height data.

9.4 Terrain Generation

Our method for visualising the processed height data in the game world can be reduced to the following steps: (1) generate a large square Unity mesh, and (2) manipulate the height of its vertices to match the received height information in real time. The bottom left of Figure 11 illustrates the key aspects of our terrain generation method.

As shown, we employ a *MapManager* to abstract away the inner workings of the system and expose key map configuration settings, as well as methods for accessing map data, such as the height or surface normal of the terrain at some world position. As part of this abstraction, we store the two-dimensional array of floats that serves as our finalised height map in the MapManager. This is passed by reference to either a Kinect interface, which receives and processes (as detailed in Section 9.3) height data from the Kinect, or a noise generator (Section 9.2.1) if the Kinect is unavailable. This minimises memory usage and speeds up reading and writing height information, compared to copying the height map between processes every time it is needed. The MapManager also divides a given map area into a configurable-size grid of *chunks*. Each chunk is passed down a unique index based on its grid coordinate, a chunk size, and a level of detail.

9.4.1 Mesh Construction

Each chunk constructs its own mesh when instantiated. This only needs to happen once, as the mesh's vertex information can be manipulated on subsequent frames (detailed in Section 9.4.2). First, we create a grid of vertices (represented as Unity `Vector3`s) inside a two-dimensional loop across the width and height of the chunk. In each iteration, we create a vertex with x- and z-values equal to the loop count plus an X- and Z-offset, which is based upon the chunk's index, and a y-value of zero. Then, we generate a list of vertex indices,

ordered such that every three indices represent a triangle. The vertex and triangle lists are then applied to a new Unity mesh, making a flat plane visible in the game.

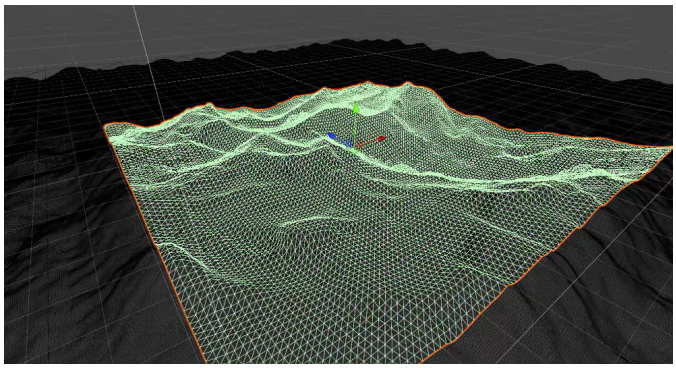


Fig. 17. Wireframe mesh of a map 'chunk', showing individual triangles

Levels of Detail

We can vary the level of detail (LOD) of a chunk mesh by dividing the original number of vertices along each side by a *LOD factor*, while spacing them apart by the same factor so that each chunk fills the same world-space area. We calculate the LOD factor by multiplying the given LOD of a mesh ($n \in \mathbb{N}$) by 2, with the exception of LOD 0, which necessitates a factor of 1. Note that the number of vertices along one side of the mesh must be divisible by the chosen LOD factor so that there are no visual artefacts due to misaligned vertices.

A benefit of dividing the map into a grid of smaller chunks is that each chunk can have its own LOD. Initially, this was designed to benefit a first-person gameplay loop, where distant chunks are rendered at a lower level of detail to improve both GPU and CPU performance. We reuse this idea to render *background chunks* at a lower resolution.
Background chunks surround the virtual play area and do not adjust their vertex heights over the course of the game. Instead, they render static Perlin noise, representing the surrounding desert, which is not under the player's control. One difficult task is lining up the edges of the static background chunks with the live play area. To solve this, we interpolate between the height of the background chunk at some *interpolation offset* from the edge, and the height of the play-area edge. This results in a smooth slope joining the two together. We then reapply the sampled noise on top of this interpolated height to ensure the seamless transition between the player's destruction and expansive desert.

Collision Handling

As we aim for our enemies to interact with the terrain, we make use of Unity's *MeshCollider* component to facilitate collisions between the terrain and other physics bodies. However, for a mesh to be usable with a mesh collider, the Unity physics system must 'bake' it by creating the spatial search acceleration structures necessary for speeding up physics calculations [9]. This expensive operation occurs whenever the mesh's data is changed, which in our case is whenever the Kinect captures a new image.
We minimise the performance impact of this mandatory operation by utilising our LOD system to generate two meshes at different levels of detail for each chunk. We then pass the lower resolution mesh to the mesh collider. This greatly reduces the time spent baking meshes on the main thread, as the number of vertices to process is decreased. We find that a LOD of 4 for the collider mesh (meaning a reduction in vertex count by a factor of 64) balances both performance improvements and matching the underlying terrain well.

9.4.2 Vertex Manipulation

To manipulate the heights of each vertex, chunks must read from the MapManager's height map. We use the chunk's index and size to assign them the correct region of the height map. The LOD factor dictates the sample period of each row of this sub-region, such that the resolutions of the mesh and this subset of the height map are the same. With the above in place, each chunk loops through its mesh's vertex list, setting the y-position equal to the appropriate height map value.

To ensure that chunks only update when new height data is available, we implement a system where the Kinect interface calls an `onHeightUpdate()` method on the MapManager once it has fully written a new height map. This method loops through all of the chunks and tells them to modify a `struct` of saved mesh data to match the new heights (shown by the arrow from the MapManager to Chunk(s) in Figure 11). A flag is then set in the chunks that instructs them to apply their saved mesh data to their Unity mesh on the next main thread `Update()`. As this is performed on the background (*Kinect*) thread, modifying mesh data for the next frame can happen in parallel with mesh baking for the current frame.

9.5 Hand Tracking & Reconstruction

From an early stage of user testing feedback, the concern of the disconnect between the real and virtual was brought up a number of times. Whilst some found it easy to relate positions between them, others struggled to pinpoint where they were interacting. To reduce this disconnect, it was decided to implement virtual hands into the game. From existing research in this area, there are a number of different options available [11], each with advantages and drawbacks.

The first distinction is between optical and non-optical based approaches; non-optical approaches often revolve around the use of wearable devices to track the position and poses of the hand, whereas optical approaches are done through cameras independent from the hand. Whilst non-optical approaches often yield more accurate results, we decided that they did not fit well with the project due to not suffering from occlusion issues and less reliance on position prediction. Mostly due to the high risk of sand damaging any wearable technology, and also the introduction of wearables takes away from

the seamless interaction with the medium, which has been important throughout our project. It also seemed sensible from a development perspective to use optical methods, as we already have access to a calibrated camera on the Kinect.

9.5.1 Technical Breakdown

Having already used MediaPipe's hand landmarks for masking out the player's hands for the terrain generation process, we now have them available in the environment. According to the documentation [12] the 21 landmarks are expressed as 3D coordinates, where the z axis represents the predicted depth relative to the wrist landmark. Using these and the associated depth image, we are able to get an accurate depth for the wrist and then extrapolate the rough height of the other landmarks. There is naturally a degree of error in this method, both in the predictions made by Media Pipe and issues caused by occlusion in the depth image, but generally it is accurate enough across multiple frames.

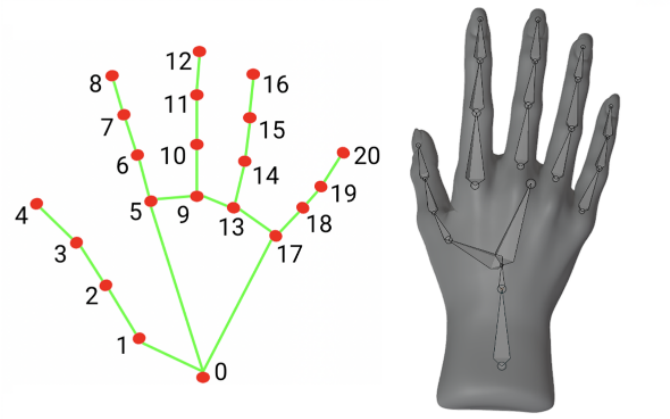


Fig. 18. MediaPipe's Landmark diagram (Left) compared to armature and model (right) being used to reconstruct the player's hands into the game.

Then, we create vectors between different landmarks using these coordinates to calculate the relative rotations of bones in our armature. This Kinematic approach is similar to techniques used in robotic hands and has the advantage of being more lightweight in comparison to other methods, such as MANO [13]. The downside to this approach is that, even with a careful degree of freedom limiting, there can still be discrepancies in the reconstruction, especially when target landmarks are inaccurate, which is common with MediaPipe. Whilst this was rarely noticeable during the normal course of the game, it was a lot clearer when tested outside of the game's interaction.
For time constraint reasons, we accepted the limitations and moved on; however, if we were to work on improving it, there are several different approaches. The main one would be to do a process similar to MANO, in which we use existing data sets on hand positions to predict the attempted hand pose and then combine this with our existing implementation to improve accuracy. Whilst MediaPipe will already do elements of this when making its landmark position predictions, we could combine this with the depth data we are receiving from the Kinect to improve these estimations. We could also switch entirely to a MANO approach of using the absolute positions of the landmarks for bones and then predicting the skin deformation based on prediction models; this could result in more natural skin deformation, but would likely need optimising heavily to maintain responsiveness.

Fig. 19. Final hand model in the game with shader applied.

9.6 Performance Optimisations

9.6.1 Inter-Process Communication

Efficient communication between the main application (running in the Unity game engine) and the separate Python process handling hand tracking was crucial for responsiveness. We initially researched several Inter-Process Communication (IPC) methods, settling on a socket-based approach using TCP. This involved sending height data from the simulated environment to the Python process and receiving hand landmark data back. However, testing revealed that this method introduced significant latency, resulting in noticeable delays in the sand updates reacting to hand movements. The performance was deemed insufficient for a real-time interactive experience.

To address this bottleneck, we transitioned to using a shared memory model. This approach allows both processes to access the same block of memory directly, eliminating much of the overhead associated with network sockets. We implemented this using the Win32 API on Windows and POSIX IPC on Linux. To ensure synchronised access to the shared memory segments, we employed semaphores. This prevented race conditions where one process might read data before the other had finished writing it, or vice versa.

The specific implementation involved Unity writing the height data from the Kinect to a shared memory segment, signalling completion via a semaphore. The Python process would wait for this signal before reading the data. Conversely, after processing, the Python process wrote the calculated hand landmark data to another shared segment and signalled via its semaphore, allowing Unity to read the results safely. This synchronised use of shared memory resulted in a significant performance improvement, ultimately yielding faster data transfer and much more responsive terrain and hand model updates.

9.6.2 Model Inference

The MediaPipe hand tracking model, while powerful, demands considerable computational resources. Running the model inference consumes a significant portion of available CPU cycles. Our application already utilises parallelisation techniques in other areas, such as map generation, to maximise performance. Consequently, attempting to run the model inference purely on the CPU, particularly within the Windows environment, led to resource contention. This manifested as lower update rates from the Python process, negatively impacting the real-time interaction quality as the hand tracking lagged behind the user's actual movements.

We investigated leveraging GPU acceleration to offload the model inference and free up CPU resources. However, we discovered that MediaPipe did not readily support GPU acceleration on Windows. This limitation was a primary motivator for exploring alternative operating systems.

9.6.3 Linux

To enable GPU-accelerated model inference, we migrated the development environment from Windows to Linux. This allowed MediaPipe to use TensorFlow to perform GPU inference on video cards that support CUDA. However, this migration posed several technical challenges. A substantial portion of the codebase and third-party dependencies were Windows-specific. For instance, the Wwise audio engine we used for spatial audio within the project does not provide a compatible shared library (DLL) for Linux, which required us to disable the Wwise editor and the associated audio features on the Linux build.

Furthermore, the rendering backend required modification on Linux. The project originally used OpenGLCore as the graphics API, which broke the rendering of shadows and some of our shaders. To solve this, we opted to switch to Vulkan, which solved the rendering issues.

Another significant challenge arose from the need to implement the POSIX IPC mechanisms (shared memory and semaphores) within our C# codebase running on Linux. The .NET runtime on Linux does not provide built-in wrappers for the standard C library (`libc`) functions required for these operations, such as `shm_open`, `mmap`, `sem_open`, `sem_post`, and `sem_wait`.

To overcome this, we utilised Platform Invocation Services (P/Invoke) via the `DllImport` attribute in C#. This allowed us to directly declare and call the necessary functions from `libc`. We created a C# wrapper class (`LinuxIPC`) that encapsulated these external function calls, providing a managed interface for creating, mapping, and synchronising access to shared memory segments and semaphores, mirroring the functionality we achieved using the Win32 API on Windows. This approach enabled seamless cross-platform IPC despite the lack of native .NET support for POSIX IPC primitives.

9.7 Enemy Implementation

Our enemy system consists of a singleton enemy manager and instances of different types of enemies that spawn in waves. Each enemy navigates through the terrain independently, taking the shortest path towards the player's stronghold.

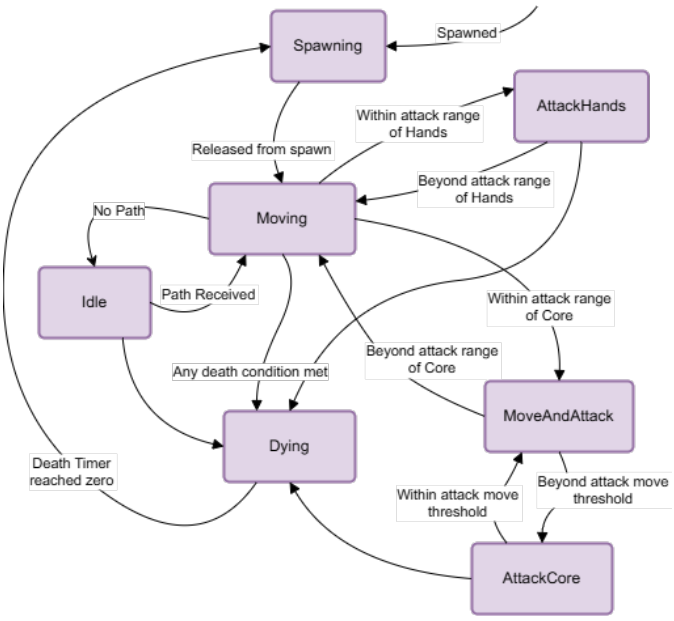


Fig. 20. Diagram displaying all enemy states and their transitions.

Broadly, enemies are implemented as Finite State Machines (FSMs). Figure 20 describes all possible states and their transitions. States are transitioned based on their position in the world relative to the Godly Core or in-game hand model, as well as specific internal triggers. This system was chosen because it allows for complete control over the behaviour of enemies at any point in time in a concise and easy-to-understand manner. The benefits of this method also extend to debugging, since we can query the state of the misbehaving enemy to quickly resolve issues.

Prompted by concerns of performance hits when large numbers of enemies are spawning, particularly in later waves,

we make use of an *enemy pool*. When an enemy dies, a reference to it is added to a queue and deactivated and reset instead of being destroyed. This allows us to reuse the same enemy instance for a future spawn, in which case we dequeue that instance. This drastically reduces the number of new enemies created and destroyed, therefore relieving CPU load and improving performance.

Below we discuss a few interesting areas of development: enemy-sand collision physics (Section 9.7.1), pathfinding (Section 9.7.2) and enemy projectiles (Section 9.7.3).

9.7.1 Physics

Guided by the game's cartoon aesthetic, the enemy physics system was designed to emphasise exaggerated, non-realistic motion. This approach prioritised low latency, high-impact feedback and physics-driven comedy. These key design details shaped three major technical features: custom death mechanics, random enemy propulsion governed by a Laplace distribution, and unique enemy interactions with the deformable terrain. As mentioned in Section 9.4, dynamically baking the colliders is computationally expensive, hindering real-time performance. As a result, custom physics solutions were implemented to supersede Unity's built-in Physics engine to better suit the atmosphere of the game.

Death Mechanics

For the majority of enemy types, there are two primary ways by which they can be defeated. The first method involves burying enemies beneath the sand. This is possible as changes to the terrain height are implemented by instantly repositioning the surface to a new elevation, rather than through a gradual physical movement. As a result, enemies may appear to phase through the terrain. To detect when an enemy has been buried, the system utilises the map manager to read the terrain height at the centre point of the enemy object and compares this value to the object's own vertical (y-axis) position. Since different enemies possess varying heights, a serialised height offset is introduced for each enemy type to ensure accurate comparison between the enemy's y-coordinate and the map height.

The second primary method of defeating an enemy is through fall damage, which occurs when the sand beneath them disappears abruptly, causing them to fall from a height. Various strategies were considered for implementing fall damage, and ultimately, a solution was adopted that combines the enemy's negative y-velocity with the map height check and a grounded offset. This approach avoids the computational expense associated with frequent collider checks or raycasts. Furthermore, using velocity to assess fall severity, rather than maintaining a history of previous terrain heights, minimises memory usage, reduces system latency, and allows for more accurate differentiation between a fall being cut short by a sudden change in the map height and otherwise.

Random Propulsion

To improve the feeling of killing an enemy, a few improvements were made to the physics system to make the interaction more satisfying. The main change was artificially increasing the chance that enemies would be launched into the air due to the movement of the sand, instead of being buried. This was based on testing feedback, where people liked the cartoon aesthetic of enemies being thrown into the air and also liked to feel powerful, as the human mind can link the idea of their hand moving in the sandpit to applying a substantial force onto an enemy, which reinforces further the immersion of the players.

Thus, we created a system based on a Laplace distribution (Figure 21) to create a random chance for enemies to be propelled. Initially, we sample the distribution to obtain a value. If this value fits within a range, the enemy is then propelled. After another round of testing, it was found that the enemies going straight up in the air was slightly boring, and while it improved the look and feel of the game, it could be much better. This then gave rise to enemies also being given a random direction in which they are propelled. To obtain the direction of propulsion, we use the sampled value and calculate the probability density of that value to get a second coordinate that would then be used as a point from the centre of this distribution's graph to create an angle that the enemy will be propelled by. This process is 2-dimensional in nature, so it is repeated to obtain an angle for the z-axis as well, which, using linear algebra, we combine into a direction vector that the enemy will be propelled.

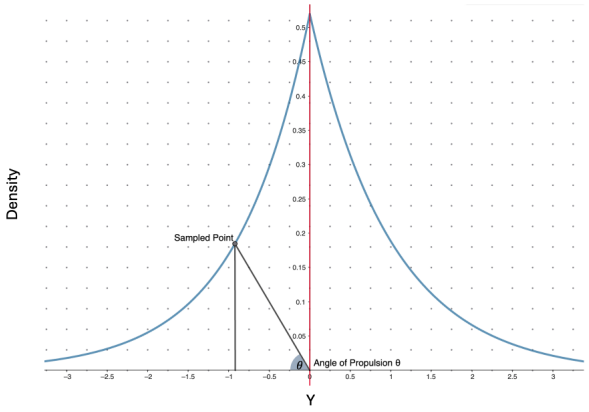


Fig. 21. Diagram showing the Laplace distribution with annotations on how it is used for vector calculations.

Novel Interactions

While numerous enemies possess unique interactions with the game environment, many are fundamentally straightforward. However, the physics-based response to vehicle explosions serves as a key example of enhancing player feedback. To create a more substantial and kinaesthetically satisfying effect upon vehicle destruction, an area-of-effect (AoE) explosive force was developed, designed to simulate the concussive

impact of a blast wave propagating outwards from the destroyed vehicle. To implement this, a `SphereCast` physics query originates from the explosion's epicentre. This query efficiently identifies all enemy colliders within a predefined blast radius (R_{blast}). Although minimising computationally intensive physics queries was a major performance concern throughout development, the `SphereCast` was deemed the optimal solution for reliably detecting all affected enemies and directly accessing their associated `Rigidbody` components necessary for force application, thus avoiding secondary data lookup.

Initially, the force applied to each affected enemy was calculated as a purely radial vector directed away from the blast centre. However, testing indicated this method produced a disappointing visual impact. Consequently, the force application logic was refined: a constant upward vector component (F_{up}) is now added to the calculated radial force vector (F_{radial}) for each affected enemy. This results in the final force equation:

$$F_{\text{final}} = F_{\text{radial}} + F_{\text{up}} \tag{5}$$

This ensures that enemies are consistently propelled both outwards and upwards, significantly enhancing the visual feedback and perceived power of the explosion, representing a deliberate prioritisation of gameplay feel over strict physical realism.

9.7.2 Pathfinding

Cursory research into video-game pathfinding reveals the A* algorithm as by far the most popular method. It also checks one big box for us: flexibility. A* uses heuristics to guide potential paths towards the goal, which may be different per enemy type, meaning enemies can have different movement characteristics that the player can visually recognise.

To implement a typical A* algorithm, we need a representation of the world as a weighted graph. This is created when the game starts up as a two-dimensional array of Nodes. Each node holds an index, world position, G-Score (start to node cost), H-Score (node to goal cost), parent node, and neighbour indices. While running, the algorithm updates the G-Score, H-Score and parent node of nodes it explores, returning a reconstructed path from parent nodes when it reaches the maximum depth limit or evaluates the goal node.

Initially, our pathfinding system ran on the main Unity thread and ended up freezing gameplay while paths were being calculated. To avoid this, we impose a producer-consumer pattern between the enemies and the pathfinder. The pathfinder exists on a background thread, continuously trying to process *PathRequests* structs from a ConcurrentQueue. Whenever an enemy is without a valid path, or after a customisable pathfinding delay, it tells the EnemyManager to add a PathRequest to the queue. A PathRequest contains a start point, an end point, and a callback method, which is called when the path has been calculated. This callback is used to assign the path to the enemy that requested it, and transition them from the Idle state back to Moving (as per Figure 20).

Common A* implementations use a priority queue to keep a list of visited nodes that have not yet been evaluated. Unfortunately, the .NET standard that Unity's libraries are based on does not include a priority queue implementation, so we are required to look elsewhere. We settle on a simple list of nodes that is sorted whenever a new element is added. This is not nearly as efficient as implementing a priority queue or heap structure ourselves, but it performs well enough for our purposes.

To aid in testing and debugging of the pathfinding system, we visualise each enemy's path in the scene view using Unity's Gizmos system. This makes it clear when a heuristic is working well, or an issue is arising, such as enemies not receiving a path at all. Figure 22 shows this in action.

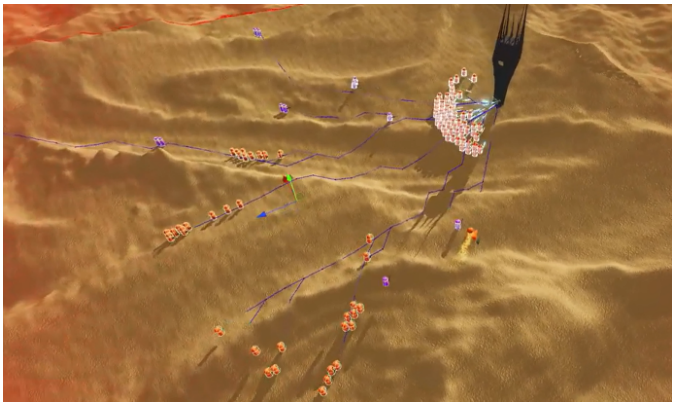


Fig. 22. Gizmos showing enemy path finding, based on A* Algorithm, towards the player's core.

9.7.3 Gunfire and Projectiles

Initially, all gunfire effects were created using Unity's ParticleSystem, and damage was calculated independently. This made it difficult to line up the visual cue of the particle hitting the core with when damage was actually applied. From internal testing and feedback from formal user tests, it was evident that this system didn't make the interaction between the enemies and the player's core clear enough.

In light of this, we implement a projectile-based gunfire system, where each "bullet" is instantiated as an object in the world. The simplest way to move the projectiles is to give them a target position and move them towards that position every frame; when the projectiles reach their destination, they apply their damage and are destroyed. Implementing them this way means that the behaviour of projectiles is consistent and predictable, without relying on built-in physics collisions or gravity simulation.

One exception to this is with the Mortar Tanks' shells, which are dealt with separately: a Rigidbody component is added to each projectile to let Unity's physics system apply acceleration

due to gravity every frame, and an initial velocity is imparted on them based upon the distance to the core. This velocity is calculated by:

$$\begin{aligned} v &= \sqrt{\frac{-|p_1 - p_0| \cdot g}{\sin 2\theta}} \\ v_f &= v \cos\theta \\ v_u &= v \sin\theta, \end{aligned} \tag{6}$$

where v_f is the velocity in the tank's forward direction, v_u is the velocity in the world space up direction, $\mathbf{p}_0$ and $\mathbf{p}_1$ are the positions of the tank and the target respectively, g is acceleration due to gravity and θ is the incline angle of the tank's barrel.
This distinction is made to improve the look and feel of the game, as well as to make it clearer what the Mortar Tanks' role is: long-distance enemies with high damage.

Due to concerns with the performance impact of creating and destroying many GameObjects, we implement a projectile pool in the same way as the enemy pool (Section 9.7), meaning new projectiles are only created if none are available to be reused.

9.8 Visuals

9.8.1 Shaders

To make our game more visually cohesive, we use Unity's shader graphs to create a number of custom shaders, most notably our *toon* or *cel* shader. This shader works by calculating lighting using the Blinn-Phong reflection model [14], converting the computed colour to the HSV colour space, and mapping the value of that colour to two or three distinct bands using a configurable greyscale texture. It is then re-multiplied by the object's albedo colour, resulting in explicit regions of light and shade, as if drawn in a cartoon. This works well with our exaggerated enemy physics system to make the game feel fun and light-hearted.

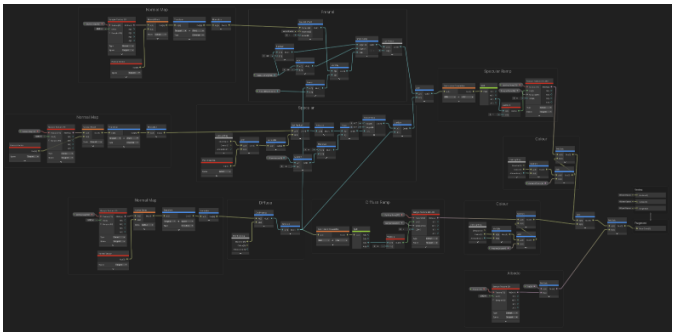


Fig. 23. Example shader graph: Toon shader used for enemy and object materials.

9.8.2 Visual Effects

Particle effects are used throughout our game to act as a fast relay of information to the player in an aesthetic way. Most of our effects are created in Unity's VFX graphs [15], building from the default templates to create custom stylised particle systems. This is done by building up layers of VFX nodes to create a final effect.

For example, the core's effect was composed of a ring of spheres that rotate around a central axis, dependent on the delta-time of the effect. These orbs each follow a periodical flickering function, where their alpha value transitions slowly between 0 and 1. This is finally combined with a small particle system for each orb to make a fireworks-like effect once the orb fades out.

Following similar design principles, we can design cartoon explosions, smoke and drone trails, to expand the look and feel of the game.

9.8.3 Newsfeed Display

Part of the audience viewing experience for the game is being able to watch the fictional newsfeed displaying the state of the game from the enemy soldiers' perspective. Presenting it as a newsfeed allows for us to apply a darker spin to the actions of the player whilst giving a more engaging experience to onlookers waiting to play the game themselves.

The system uses a set of render textures that get attached to cameras, which are switched between holder points across the map. These exist as "news helicopters" allowing sweeping views of the battlefield, and enemy-holder points, which are created on a percentage of spawned enemies. Using a subscriber model, they are then registered to the camera controller and can now be initialised in the newsfeed.

9.8.4 Post-Processing

To create a distinction between the game visuals and the newsfeed visuals, post-processing effects are applied to each set of cameras for different purposes. The game camera has its colour rebalanced to be more cartoony and vivid, the purpose of this is to make the game more straightforward for the player and give greater visual depth in the sand.

The newsfeed camera is conversely tone shifted down, with more saturated colours and artificial film grain added. This is to recreate a more photo-realistic aesthetic, which aims to elicit a darker theme to contrast with the more light-hearted nature of the actual game. This is achieved using Unity's render pipeline by applying separate volumes [16] to different virtual cameras in the game space.

9.9 Discontinued Systems

9.9.1 Networking

Initial visions for the game saw the Kinect data being streamed to clients playing an FPS on the moving 3D terrain mesh. This required a client/server model, which was implemented using Fishnet Multiplayer [17], a lightweight Netcode package that offered improvements on Unity's base functionality, whilst still retaining flexible deployment options.

Fig. 24. Post Processing effects applied to game cameras to create different effects.

Figure 25 shows how data was exchanged between the server machine physically connected to the Kinect and any number of clients running the game.

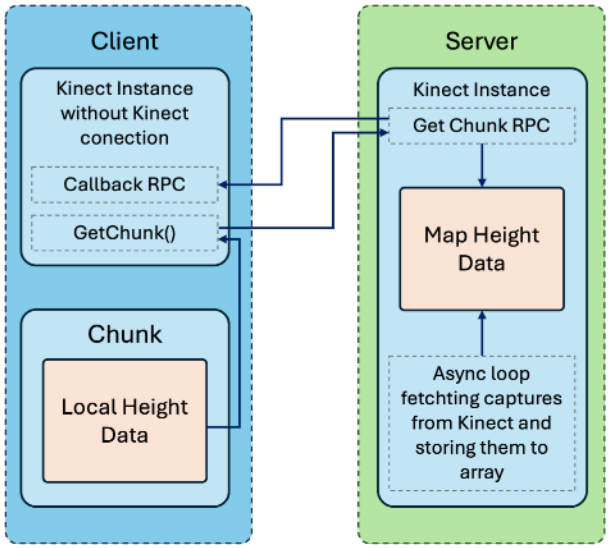


Fig. 25. Diagram showing how the exchange of Kinect data was handled using RPC between client and server.

Performance of the multiplayer implementation was a huge limiting factor in the overall feel of the game, due to the large amount of data being streamed across the network, the game experienced slowdowns and frequent desyncs. This, paired with the lack of interest in the FPS aspects of the game, led to the major pivot in the vision of the final game.

9.9.2 First-person Movement & Physics

Before the pivot, the vision for the game looked very different. At the time, the game was split into asymmetric interactions aimed at multiple players, with some playing a first-person shooter (FPS) on a map, controlled by a "gamemaster" playing in the sandbox.

The movement system that we had envisioned for that was expansive, detailed and aimed at providing fluid motion on the ever-shifting terrain. Alongside standard movement, such as jumping, crouching and sprinting, there were also more dynamic movement options, such as sliding, wall riding and concepts of momentum conservation. Taking inspiration from games like Titanfall 2 [18], the aim was to build a fast-paced and fluid system that allowed players to respond to the changing requirements of the terrain.

This was implemented as an enumerated state switching system, which allowed storing the player's complex state. Movement was then composed of applications of force onto the player's Rigidbody. Whilst this worked well in testing, it encountered some issues when interacting with the moving terrain mesh:

1) The most obvious is that our game was going to take place inside a sandbox, which results in lots of slopes and hills in the recreation, and so applications of force need to be adjusted to respond to them. By using a raycast to check the distance to the ground, we can use the collision to find the normal to the ground beneath the player and give some bounding limits to how steep the slope must be to change its effect on the player's movements. Instead of applying a force into the slope directly forward, adjusting the direction of the force to follow the slope, stopped the player from clipping, bouncing and getting stuck on slopes.
2) The trickiest feature to implement was the wall running and jumping, as it relied on linear algebra calculations about the orientation of the camera and player, as well as the input direction of movement. For this, another raycasting solution was used, and this meant that there were distinct differences in running forward, backwards or up a wall. This was done using 3 raycasts forward, left and right, and keeping a hit boolean of which ones hit and which don't, and then relaying that to the current movement of the player. The difficulty comes when a player changes their looking direction whilst on a wall. Keeping these hits up to date allowed the script to keep the player's momentum going, even though they are turning away from the direction of movement.
3) The last hurdle with wall running was how to properly scale a wall. What happens is you get to the top and get stuck there with no way to go. The solution was a mantle system which detected if you are close enough to an edge using another raycast from a point a small distance above the player's head, and then pushing the player out and up over the wall.

9.9.3 First-person Weapon System

An initial prototype for the first-person weapon system was developed using the FishNet networking library. This implementation employed a server-authoritative model, specifically for hit detection, to maintain game state integrity and mitigate cheating. While the server processed and validated all hit registration events originating from weapon

fire, the responsibility for triggering and displaying visual and audio feedback was delegated to the client, aiming to enhance perceived responsiveness for the player.

The system incorporated core functionalities expected of a modern shooter, including weapon swapping capabilities, allowing players to cycle through their equipped weapons. A comprehensive ammunition management system was implemented, tracking both the current magazine count and the total reserve ammunition for each weapon. This included logic for player-initiated reloading sequences, which would replenish the current magazine from the total ammo pool, contingent on available reserves.

Client-side effects were implemented to provide immediate feedback to the firing player. Audio cues included distinct sounds for weapon firing, reloading actions, and an "out of ammo" notification. Visual effects primarily focused on muzzle flashes and bullet tracer effects, created using Unity's default particle system package. Despite incorporating these features, this particular implementation was ultimately deprecated and did not proceed to the final version of the project due to the pivot.

10 Acknowledgements

The development of this project has received support from a number of individuals, without which we could not have fulfilled this undertaking. In no specific order, we want to extend thanks to Robbie Devine and Sam Evans for creating incredible music and sound effects for the game. Amy Bland, for allowing us to use the BIG lab for initial development. James Filbin for assisting with the Sandbox construction. Alex Elwood, Jed Preist, Aaron Zhang and George Sains, for meeting with us weekly to give invaluable feedback on the project and give academic advice. Sarah Connolly, for green-lighting the project and supporting us throughout. Luke Woodbury and the rest of the technical support team for providing equipment, and to everyone else who has assisted with testing, provided advice or helped in development.

11 References

[1] F. Wellmann, S. Virgo, D. Escallon, M. de la Varga, A. Jüstel, F. M. Wagner, J. Kowalski, H. Zhao, R. Fehling, and Q. Chen, "Open AR-Sandbox: A haptic interface for geoscience education and outreach," *Geosphere*, vol. 18, no. 2, pp. 732–749, 02 2022. [Online]. Available: https://doi.org/10.1130/GES02455.1

[2] GameCI, "GameCI documentation," https://game.ci/docs/, accessed: 25 April 2025.

[3] Unity Technologies, "Unity profiler documentation," https://docs.unity3d.com/Manual/Profiler.html, accessed: 22 April 2025.

[4] Microsoft, "Azure Kinect DK," https://pub-c2c1d9230f0b4abb9b0d2d95e06fd4ef.r2.dev/2019/06/Factsheet-Azure-Kinect-DK.pdf, accessed: 22 April 2025.

[5] T. de Monterrey, "Azure Kinect DK specifications (skeletal tracking)," https://ifelldh.tec.mx/sites/g/files/vgjovo1101/files/Azure%20Kinect%20DK%20Specifications.pdf, accessed: 22 April 2025.

[6] Microsoft, "Azure Kinect sensor documentation," https://microsoft.github.io/Azure-Kinect-Sensor-SDK/master/index.html, accessed: 24 April 2025.

[7] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M. G. Yong, J. Lee *et al.*, "Mediapipe: A framework for building perception pipelines," *arXiv preprint arXiv:1906.08172*, 2019.

[8] R. Brown, *Smoothing, Forecasting and Prediction of Discrete Time Series*, ser. Dover Phoenix Editions. Dover Publications, 2004. [Online]. Available: https://books.google.co.uk/books?id=XXFNW_QaJYgC

[9] Unity Technologies, "Unity scripting reference," https://docs.unity3d.com/6000.0/Documentation/ScriptReference/index.html, accessed: 18 April 2025.

[10] G. Bradski, "The opencv library." *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[11] S. Jörg, Y. Ye, M. Neff, F. Mueller, and V. Zordan, "Virtual hands in VR: Motion capture, synthesis, and perception," https://www.uni-bamberg.de/fileadmin/cg/publications/joerg2020/Joerg20_SiggraphAsia_courseNotes.pdf, accessed: 21 April 2025.

[12] Google, "Documentation for mediapipe hands landmarking," https://mediapipe.readthedocs.io/en/latest/solutions/hands.html, accessed: 21 April 2025.

[13] J. Romero, D. Tzionas, and M. J. Black, "Embodied hands: Modeling and capturing hands and bodies together," pp. 1–17, 11 2017. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/3130800.3130883

[14] Wikipedia, "Blinn-Phong reflection model," https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model, accessed: 26 April 2025.

[15] Unity Technologies, "Unity VFX graphs documentation," https://unity.com/features/visual-effect-graph, accessed: 25 April 2025.

[16] ——, "Unity URP volumes documentation," https://docs.unity3d.com/6000.0/Documentation/Manual/urp/Volumes.html, accessed: 26 April 2025.

[17] FirstGearGames, "Fish-Net: Networking evolved documentation," https://fish-networking.gitbook.io/docs#overview, accessed: 24 April 2025.

[18] Respawn Entertainment, "Titanfall 2," https://store.steampowered.com/app/1237970/Titanfall_2/, accessed: 25 April 2025.

12 Appendix

12.1 More Discontinued Systems

12.1.1 Object Detection

Before we knew what our game would look like, many possible interactions with the sand and the kinect camera were proposed. One such interaction was object detection in a simmilar manner to how we work hand tracking. Either by 3D printing different shapes or QR codes onto said shapes, we could then place different objects into the world.

We had a working version of this, which we showed off in a weekly meeting and briefly during the MVP panel. In the end this never went anywhere but again shows the depth in thought, research, and technical experimentation that we underwent.

12.1.2 Hand Gestures

Using the same MediaPipe library that we already used for hand tracking and masking, we could extend the functionality of our python script to check when a select number of hand gestures were being performed in front of the camera. Some examples are “Thumbs up”, “Closed fist”, “Open palm”, etc... Another proposal for novel interactions was to create a spellcasting system where using hand gestures could activate a special spell like a “Lightning Strike” or “Meteor”.

This would add yet another layer of depth and strategy to the game. We had and still have a system which prints any hand gesture currently being used onto the screen of the python overlay (which is used only for development and testing purposes) but we found that our focus needed to close in on polishing the features and gameplay we already had to make the best possible experience, rather than stretch ourselves thin and leave some lag or unpolished features in the game.

12.1.3 Mech Final Boss

Another really interesting and technically challenging interaction with the sand that we created was a mech suit final boss which once it shoots, turns into a giant mecha-ball which you can roll around in the sand and try to run it into its own missile. This feature was very close to completion but was left without a proper model and also largely untested and so as a team we decided to not push for it into the final release of the game. The tech behind it however is still very interesting and challenging due to Unity’s innate lack of strength when it comes to simulating deforming / dynamic meshes.

A ball in unity on a perfectly static mesh, even one as rough as the map we generate, will roll very cleanly with little to no clipping and no getting stuck. However, the moment that mesh is dynamic, it will suddenly clip every frame, get stuck, not speed up or roll upwards and many more hilarious and frustrating bugs. Many solutions were put forward and tested, but the one that got the best results was: firstly, to completely supersede the collision and pin the ball to the height of the map if it falls below it. This is a very common tactic when tackling dynamic meshes. This alone solved next to nothing in our particular situation. However, as the ball now stays completely still and never rolls anywhere. By removing the collision detection for the ball, we now have a new problem, the ball has no normal contact force being applied to it by the floor. This means that when gravity pushes it down a slope, it falls down but by our script gets pinned to the height of the map. Therefore, staying perfectly in place.

To simulate the roll that would otherwise happen, we first tried to raycast to the floor to find the normal direction to the plane and then by hand simulate the force that would be applied normally. This also proved to be unstable and also quite computationally costly. So the final solution we came up with is placing 4 points at the 4 cardinal corners of the ball (NW, NE, SW, SE) and reading the map height at each of these points, then building up a direction vector based on the combined magnitude of these distances and directions. This lead us to a new direction vector we can use to roll the ball in the direction gravity would realistically roll it.

By allowing us to leave gravity on, the ball still acts naturally in the air but now also rolls in a fairly realistic manner on the ground. This was all tested and worked to an extent the team was happy with; the elements that weren’t fully tested are the death by its own missile and transformation into a ball.

12.2 Extra Pictures

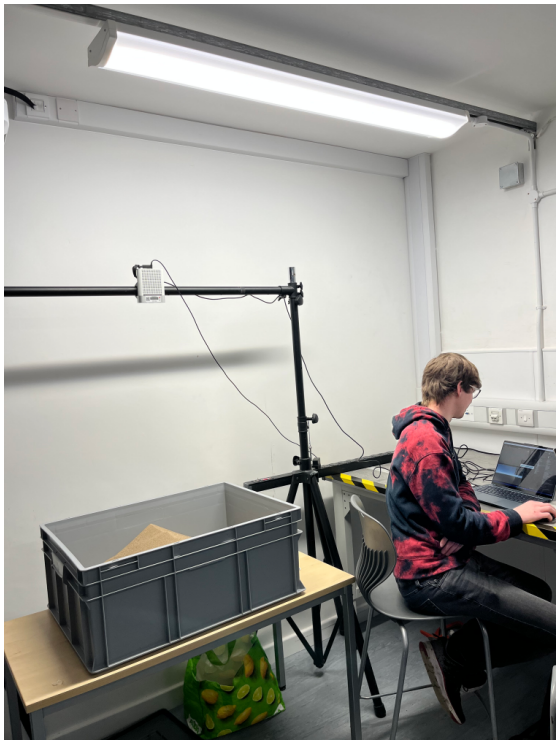


Fig. 26. Picture showing what our setup in the BIG lab looked like.

Fig. 27. Picture of team members transporting the sand from Queens to MVB for the beta panel.

Fig. 28. Picture of team members Shovelling sand into a transport box ready to move to 1.11.

Fig. 29. Another picture of team members Shovelling sand into a transport box ready to move to 1.11.

Fig. 30. Picture showing what 1.11 looked like before we set up.

Fig. 31. Picture showing what 1.11 looked like after we set up.

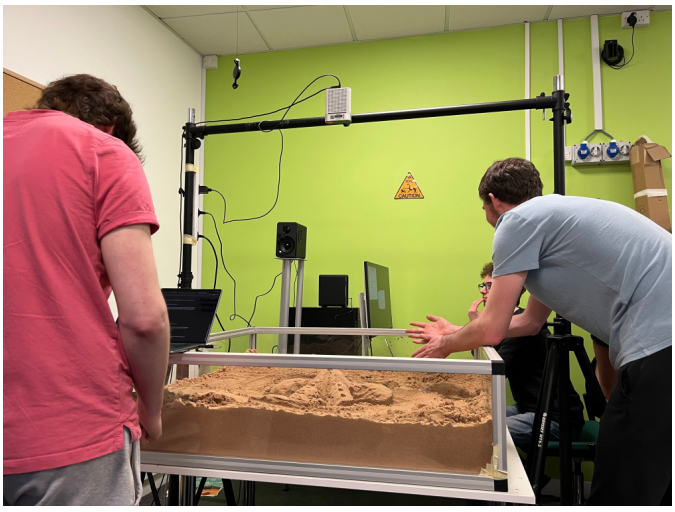


Fig. 32. Picture showing team members testing in 2.56.

Fig. 33. Picture showing what games day looked like for the public.

Fig. 34. Picture showing team members pair-programming.

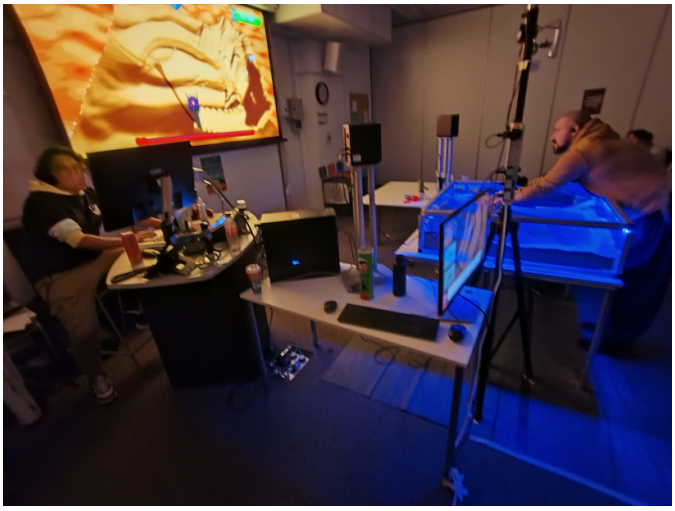


Fig. 35. Another picture showing what games day looked like for the public.

Fig. 36. Picture showing team members building the sandbox.

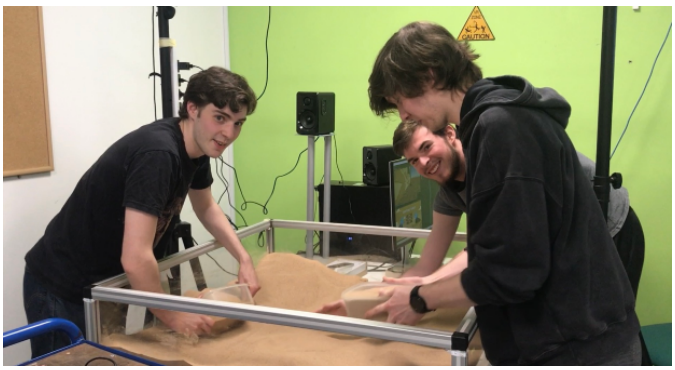


Fig. 37. Picture showing team members shifting sand for Games Day.