iT AiN'T THAT DEEP

Major Project Report

MSc Creative Digital Media & UX
School of Media, TU Dublin

Supervised by Dr. Brian Vaughan

Submitted by
Jibril Abdulazeez | D24125280
Jacob Abraham | D24125466

Trello Link: https://trello.com/b/J48LiEw0/dolphin-simulator

Store link: https://jibbyie.itch.io/it-aint-that-deep

ACKNOWLEDGEMENTS

Without the support and participation of many, this project would not have been completed on time. We truly appreciate and proudly acknowledge their gracious contribution

We want to convey our sincere gratitude to everyone, particularly to the following individuals and organizations.

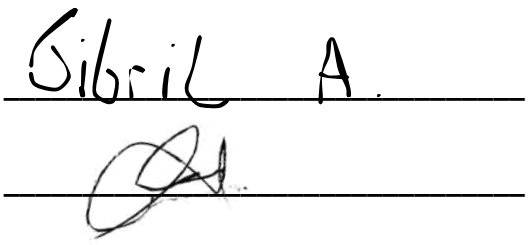
To begin with, we would like to thank Technological University Dublin for this opportunity. This experience indeed provided us with great exposure and irreplaceable knowledge. Next, we would like to thank Dr. Brian Vaughan, our supervisor in this major project, whose invaluable assistance and guidance made it possible to complete this project.

We would also like to thank Dr. Keith Gardiner and Prof. Evin McCarthy for their guidance and support throughout this venture. We are so grateful for the help and support of Ms. Daria Babaskina, who was present from day one till the launch of this game.

And last, but not least, we are grateful to all our friends, family, and loved ones, who were part of this effort and stood with us in completing this major project

DECLARATION

We, Jibril Abdulazeez and Jacob Abraham, hereby certify that the material which is submitted in this final report, towards the Award of a Master of Science in Creative Digital Media & UX, is entirely our own work and has not been submitted for any academic assessment other than part-fulfilment of the award named above.
Signed:



Date:

30th November 2025

CONTENTS

INTRODUCTION

This report details the design, development, and documented process followed till the submission of this MSc Major Project. We take a look at the creation of a simple yet sophisticated sandbox with an interesting storyline and tons of fun. This report presents user research data, competitor analysis, development methodology, testing, and evaluation. A critical analysis of all these factors that led to the final version of the vertical slice of this game is detailed in this document.

The game "iT AiN'T THAT DEEP" is a 'road not taken' venture in a world of cash cow game studios. We wanted to dial back a few years to when the purpose of a game was to have a fun time and get some dopamine flowing. While the first and foremost purpose of this report is to showcase our work and the process followed in completing this Major Project, we also wanted to show that an idea shouldn't have to be that deep if it is worked on well.

"iT AiN'T THAT DEEP" is a simple underwater first-person shooter game (just for the vertical slice), where the main character is an unnamed dolphin with a hunger for chaos and destruction. The game has a mission-based progression with pearls (money - Not Included in this vertical slice), Aura(cred), and other collectibles, which help the player progress and explore. Since the scope of such a game is too high for an indie development team, a vertical slice is being presented as the MVP.

Section 1- User Need Analysis contains the user research, which contains all the data we came across, and the secondary research leading to the finalization of the idea, leading to the final output. Following that is a very detailed persona section, along with some user scenarios.

Section 2- Background Research is tipped off by a detailed competitor analysis, followed by non-domain influences. From there, we move on to the tech stack and the definition of design requirements.

Section 3- Methodology and Approach talks about the overall design process of the game, including but not limited to UI, Environment, Character Design, Missions, and Storyline. The second part of this section takes you through the code and code workflow of this game. Closely followed by the architecture, functionality, and other snippets of development, ending the section with attributions to AI use.

Section 4- Testing & Analysis illustrates the user testing methodology, and other details, including participant data, testing environment, questionnaire, ETPs, conditions/metrics, and a detailed report on the testing sessions. Finishing up with the results and feedback, along with future recommendations.

Section 5- Evaluation comprises Critical Success Factors, Minimum Viable Product, an evaluation of Code, Design, Art, Audio, and Technical components. This section also includes the division of labour, future endeavours, Marketing, and Social Media.

USER NEED ANALYSIS

INTRODUCTION

This section deals with the core needs, motivations, and behaviours of players engaging with our game. We can get an idea of how users interact with the game and what they seek from the personas created and the simple user scenarios in this section. The section starts of with the user personas, followed by scenarios, and objectives based on user needs.

USER PERSONAS

Persona 1: Liam

- **Age:** 19
- **Occupation:** First-year student
- **Play Style:** Casual, chaotic, non-linear
- **Session Length:** 30–60 minutes
- **Favorite Games:** Goat Simulator, Untitled Goose Game, Lethal Company, TABS

Goals:

- **Quick Chotic fun-** He wants to have hilarious, unfiltered moments
- **Breaking systems-** Turning every event hostile and attacking random NPCs
- **Unintended behavior-** Loves to find glitches, exploits, and find a way to play it wrong
- **Be a memer gamer-** Find moments to share with the community

Frustrations:

- Forced tutorials
- Complex systems and heavy management
- Badly scripted jokes
- Hyper-realism

Gameplay Behaviour

- Mostly ignores the main quests
- Stress test to the threshold
- Chooses the unconventional option every time
- Replays the fun moments
- Likes to share his findings, ie spread chaos

Liam is our core player – the kind of player that wants to boot up a game to mess around and see how far he can push the systems.

Persona 2: Jess

- **Age:** 25
- **Occupation:** Twitch streamer
- **Play Style:** Expressive, chaos-driven entertainer
- **Session Length:** 1–3 hours, mostly evenings/weekends
- **Favorite Games:** Human Fall Flat, Among Us, Don’t Scream, Party Animals, Viscera Cleanup Detail

Goals

- Generate entertaining content for the viewers
- Keep the streams going seamlessly
- A lot of free play
- Chat engagement through unscripted chaos

Frustrations

- Long onboarding
- Restrictive narratives
- Poor UI, Visual clutter

Gameplay Behaviour

- Utilises NPCs for entertainment
- Amplifies chaos
- Keeps game objectives optional
- Tests game reactivity

USER SCENARIOS

Scenario 1 - Liam: Late-Night Distraction

It’s past midnight and needing a quick break, Liam opens up It Ain’t That Deep for a short, fun session. He opts to go on a mission, triggering a brief dialogue with an NPC who gives him a list of tasks. Ignoring that, he drifts in the general direction, searching for some coveted collectables, a pool stick, the lost candy earring, each giving him some extra aura. NPCs respond variably: many ignore him, others offer simple voiced lines or animations upon interaction. He maintains a low-effort session.

Scenario 2 - Jess: Evening Stream Session

It’s time for the daily stream session, Jess chooses “iT AiN’T THAT DEEP” just as the chat requested. As soon as it launches, she starts to move and interact freely with the environment, without forgetting to engage the chat. And then moves on to repeat the worst dialogue choices with the NPCs, as the viewers are loving it. She goes around the environment, testing the physics and its limits, and finding one collectible after another. An hour has passed, and she hasn't made a dent in the mission progression, but the chat is in chaos. They love the new content, a game focusing on the player and not the marketplace.

Objectives

It Ain't That Deep is a polished vertical slice of an absurd first-person shooter game. The Objective is to deliver a juiced-up version of the vertical slice by the end of the MSc project. The overall design philosophy is “structured chaos”; the system will be clean, modular and extensible, even if the gameplay looks unhinged on the surface.

SMART Objectives

Specific:

Build 5 modular gameplay systems: dolphin movement, interaction toolkit, dialogue system, basic NPC AI, Enemy AI systems, event-based trigger system. Create a mission-themed zone with its own questlines.

Measurable:

Deliver a functional prototype by June, core systems by July, content pass by September, continuous testing throughout and a final build in November, progress tracked via Trello.

Achievable:

Systems are within reach for a solo developer alongside a novice developer, using Unity and available plugins such as Pixel Crushers Dialogue Systems. No networking, multiplayer, or high-fidelity demands. Art and writing handled by the designer (Jake).

Relevant:

Aligned with MSc project goals, demonstrating gameplay programming, UX iteration and game design.

Time-Bound:

Development is broken into monthly sprints, concluding with a playable, testable, and polished demo by November.

Included Features (Core Focus)

Movement System

- Dolphin Movement in 3D space (swim, sprint, shoot, interact).
- Physics-based motion (ragdoll dives, flops, knockbacks, etc.).

Combat System

- Complete weapons system & First Person shooting.

Dialogue System

- Branching conversations.
- Rewards based on conversational state.

Enemy AI Systems

- Patrol AI for NPCs.
- Reactive states (Shoot).
- 4 Separate Enemy AI Systems - Pistol, Sniper, Melee, Boss.

Trigger & Event System

- Player interaction choices with the NPCs will increase/decrease cred or ammo
- Dialogue changes based on NPC memory/state.
- Collectibles & Aura (points).

World Design

- Small, dense underwater map (2 zones).
- Interactable NPCs, environmental triggers, and light physics props.

UI Systems

- Basic HUD with interaction prompts.
- Dialogue box and choice interface.

Excluded Features (Out of Scope)

Multiplayer/Online Play

The game is entirely single-player to avoid the complexity of networking, syncing or dedicated servers.

Options & Settings

Open World Scaling or Procedural Generation

Map is handcrafted and small in scale to allow focused interaction design.

Levelling Systems

No skill trees or XP mechanics, the goal is freeform fun and not progression loops. The only count or progression in the game is with Aura and Ammo

CONCLUSION

To meet the demands of its primary audience, the game must prioritize quick, responsive, and low-commitment gaming. Liam and Jess both prioritize spontaneity, humor, and flexibility over complication or organization. It is evident from their actions and complaints that they all want systems that reward innovation, promote chaos, and create memorable, recurring experiences that may be shared socially or enjoyed informally.

This part lays the groundwork for the game's design approach by comprehending these demands through user personas and scenario-based research. The results highlight the significance of limited onboarding, robust feedback loops, high levels of involvement, and an environment that is entertaining and non-linear. These observations guide the design choices and development techniques discussed in the next chapters, guaranteeing that the game stays in line with player expectations and motivations.

BACKGROUND RESEARCH

INTRODUCION

This section includes the competitor analysis, and the non-domain influences come next. The tech stack contains information about the chosen platform, software preferences, distribution strategy, promotion, and Research to finish this part off. This section also has the definition of design requirements, which focuses on the design decisions taken in regard to the target audience.

COMPETITOR ANALYSIS

Goat Simulator

Genre: Sandbox / Physics Comedy
Developer: Coffee Stain Studios
Platform: PC, Console, Mobile

Strengths

Unconstructed Creativity: The game gives the player the freedom to explore the map without constraints. The gameplay is emergent and addictive.
Comedic Physics: The physics of this game is set up in such a clumsy way, it brings out unintentional humor in every nook and corner.

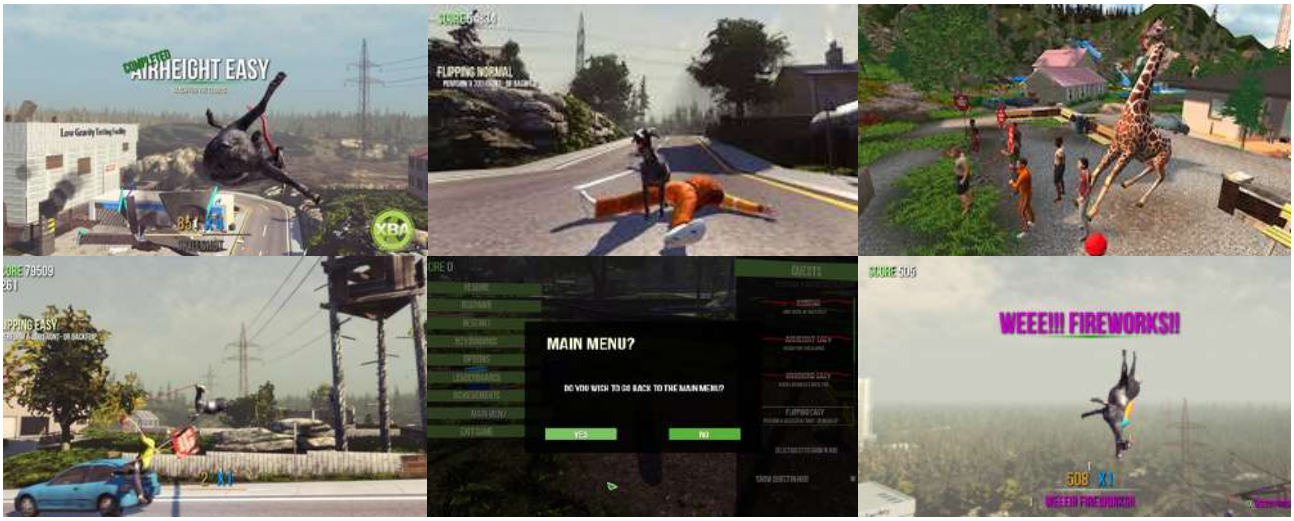
Weaknesses

Shallow Core Mechanics: The game has little to no depth in core mechanics, and that reduces user engagement.
Poor Feedback System: The game cues seem confusing and vague. The game has unusual pop-ups and lacks environmental cues.
Minimal Progression: The game has minimal to no progression; the addictive, humorous effect wears off in a while, and the game ends up being just extra space on the computer.

Inference

- The game was indeed a success and made a profit in the open market, and went on to have sequels, but the strengths aside, the game does have some major holes. In “iT AiN’T THAT DEEP”, we will address these issues and try to make it different as this is our major competitor.
- We will adopt the whole sandbox model and have the absurdity in the gameplay just like ‘Goat Simulator’. But we will have a skill-driven progression system in place that will give the players an incentive to play the game even after the effect of that first impression wears off.
- We do have a simple and small environment in place, but there is room to extend it to how much we want. The game is mission-based, and each mission has a specific area and theme.

Screenshots



UNTITLED GOOSE GAME

Genre: Puzzle-Stealth Comedy
Developer: House House
Platform: PC, Console

Strengths

- Playful Objective Design: To-do list format encourages goal-driven mischief without rigid handholding. Reactive World: NPCs and objects respond naturally to the goose's actions, encouraging experimentation.
- Streamlined Interface: Clean visuals and minimal UI help players focus on world interaction. Humour via Gameplay, Not Dialogue: Comedy emerges from the systems and timing, not scripted jokes

Weaknesses

- Short Playtime: Once objectives are completed, replay value is limited.
- Basic Movement Mechanics: Movement is responsive but simplistic and non-evolving.
- Limited Mechanical Growth: Lacks new abilities or complexity over time.

Inference

Untitled Goose Game was a success due to the originality of the idea and the simplicity of design. The only drawbacks were, in fact, the short playtime and limited progression. While "iT AiN'T THAT DEEP" follows a similar idea, we omit certain factors like a limited progression, while adding on a more skill-based progression required for a first-person game. Our game encourages unscriped chaos, rather than checklist humour that's found in this game.

Screenshots

NON-DOMAIN INFLUENCE

‘Our Planet’ - Netflix Documentary

Genre: Nature Documentary
Produced by: Netflix
Platform: Mobile App, Web

Influence Type: Storyline, Behavior

Intergration

- The behavior of dolphins in the wild is to be studied and integrated in-game.
- The environment and habitat of these animals
- Their interaction with other marine life can provide a great deal of data for NPC interaction in-game.
- Predators and prey info for creating enemies and dangers in-game.

Value

- Taking inspiration from a documentary gives a taste of realism.
- Interactions would not look like forced humour.
- The environment would be geographically accurate.

‘Family Guy’ - Animated TV Show

Genre: Animation, Dark Humor, Satire
Produced by: Fox Entertainment
Platform: Mobile, Web

I**nfluence Type:** Pop-culture References, Surrealism, Layered Satire, Dark Humour

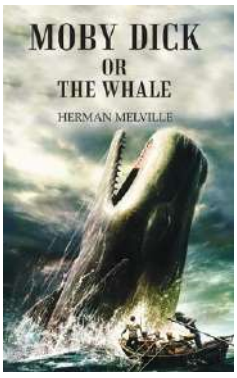
Intergration

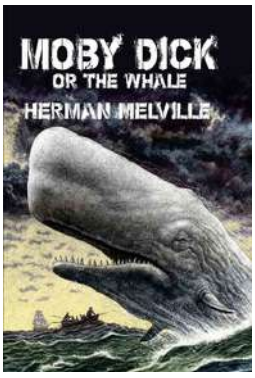
- Cutaway scenes
- Pop-culture references
- Literal satire
- Dark humour

Value

- Creates a player-led narrative pace and intensifies the absurdist tone.
- Checks the gaming cycle without reducing player immersion or agency.

‘Moby Dick’ - Book





Genre: Fiction
Written by: Herman Melville
Publisher: Harper & Brothers

Influence Type: Narrative Structure, Obsession, Symbolism

Intergration

- Environmental storytelling: Sunken whaling vessels, skeleton
- Symbolism of a rogue animal

Value

- Adds intrigue and replayable narrative complexity without making the game run more slowly.
- Encourages investigation and curiosity by contrasting chaos and depth.

TECHNOLOGIES

The weapons system had to support the project's overall philosophy of "structured chaos": the shooting needed to feel unhinged and expressive for Liam/Jess, but the underlying code had to stay modular and easy to extend.

Core technologies used

Unity (C# MonoBehaviours + ScriptableObjects)

- Each weapon is defined as a ScriptableObject (WeaponData), separating data (damage, range, ammo, VFX/SFX) from runtime behaviour. This lets us balance weapons and add new ones directly in the editor without touching code.

Event-driven architecture

- Systems subscribe to weapon events instead of hard-coding references. WeaponManager exposes a static CurrentWeapon and an OnWeaponSwitched event that UI, crosshair and state trackers listen to.

Physics raycasts & spherecasts

- Hitscan weapons (pistol/rifle) use Physics. Raycast, while the RPG uses Physics.SphereCastAll for splash damage, both driven by WeaponData.range and WeaponData.sphereCastRadius.

Layer-based hit detection & damage types

- Weapons carry a DamageType (Slap, Melee, Pistol, Rifle, Explosive), which is matched against each enemy's immunities (DamageTypeImmunities). Damage is only applied when the target is not immune, allowing designer-friendly control over which enemies respond to which weapon.

UI integration (TextMeshPro + Unity UI)

The weapon HUD is split into small, focused controllers:

- FirstPersonUIController shows weapon name, icon and ammo counts.
- WeaponStateTracker and WeaponStateUIController manage state-specific icons (idle, firing, reloading, out-of-ammo).
- CrosshairController changes reticle sprite/color based on the equipped weapon and whether a shot would actually deal damage.

Design Tools

- All the buildings were modeled and textured in Blender 3D, except the nature assets
- The 2D assets were hand-drawn in Adobe Illustrator in vector format to maintain quality and then exported in an 8K (7680×4320 pixels) resolution PNG
- The UI component was created in Adobe Illustrator to achieve the depth needed for game UI and then prototyped using Figma

Target Platform: Windows PC
The primary build target is **Windows PC**. This was chosen for tworeasons:

1. **Development environment alignment** – All project tooling (Unity editor, version control, profiling tools) was already set up on Windows machines, reducing configuration overhead and friction during debugging.
2. **Audience reach** – Windows remains the dominant platform for PC gaming, with strong support for gamepads, mice, high-refresh monitors, and audio hardware. Targeting Windows first maximises the chance that playtesters can run the build without extra setup.

Game Engine: Unity
We chose **Unity** as the core engine for this project for both practical and pedagogical reasons:

- **Extensive documentation and community** – Unity has comprehensive official docs, tutorials, and a large community. When implementing more advanced features (e.g. hit-stop, camera kick, ScriptableObject-driven weapons), We could cross-reference multiple examples instead of working from first principles.
- **Asset Store ecosystem** – The Unity Asset Store provides a large catalogue of **free and low-cost assets** (models, textures, shaders, editor tools). This allowed me to focus my time on gameplay code and bespoke systems while still achieving acceptable visual and audio quality.

Flexible build pipeline – Unity's build settings make it straightforward to target multiple platforms (Windows, WebGL, etc.) from the same project. This is important for both local testing and eventual distribution.

Engine Version: Unity 6
The project uses Unity 6 (the current LTS generation at the time of development). This version was chosen instead of an older LTS for a couple of reasons:

- Better editor UX and workflows – The newer editor version improves prefab workflows, search, and package management, all of which reduce friction when iterating on weapons, UI canvases, and enemy prefabs.
- Forward compatibility – Targeting Unity 6 means the project is aligned with the engine version that will be supported for the next few years, which is important if the game is extended post-submission.

Software Stack & Tooling
The core software and tools used in the project are:

- Unity 6 Editor – Main development environment for scenes, prefabs, lighting, and build management.
- C# (Unity scripting) – All gameplay systems (weapons, health, trickshot tracking, aura, UI) are implemented in C# scripts.
- IDE (VS Code) – Used for editing and debugging C# code with IntelliSense, breakpoints, and refactoring.
- Git / Version Control – Used to manage project history, experiment with branches for risky changes, and provide a clear timeline of iterations.
- Unity Package Manager – For pulling in official packages (e.g. TextMeshPro, post-processing) and keeping them consistent across machines.

Each tool was selected to minimise friction: everything integrates directly into the Unity workflow, keeping context switches low during rapid iteration.

DEFINITION OF DESIGN REQUIREMENTS

Design requirements for the weapons system

From the user personas and design goals, the weapons system had to:

- **Support multiple weapon classes** - Melee (slap, pool cue), pistol, rifle and RPG had to co-exist, each with its own damage, range and reload behaviour.

- **Be swappable and readable at a glance** - Players needed to switch weapons quickly using number keys or the mouse wheel, with instant audio feedback and HUD updates.

- **Expose "skill-based chaos" through ammo and UI** - Ammo tracking needed to be clear but generous: hitting shots and finding ammo drops should keep players in flow instead of punishing them with micro-management.

- **Be extensible for future missions** - Adding a new gun should be mostly data entry (creating a new WeaponData asset and plugging it into WeaponManager and UI mappings), not rewriting firing logic.

- **Respect global systems -** All weapons must obey the pause state (PauseMenuController.IsPaused) so no input is processed while the game is paused.

Design requirements for the weapons system

From the user personas and design goals, the UI and Visual details had to:

Be simple and casual - The UI must give a playful feel rather than a horror or sombre look of its counterparts.

Reduce Onboarding time - The UX of this game is built in such a way, the gameplay comes first.

Chaos first style - The art style takes you back a few years, where games were simple and made for fun

METHODOLOGY AND APPROACH

INTRODUCTION

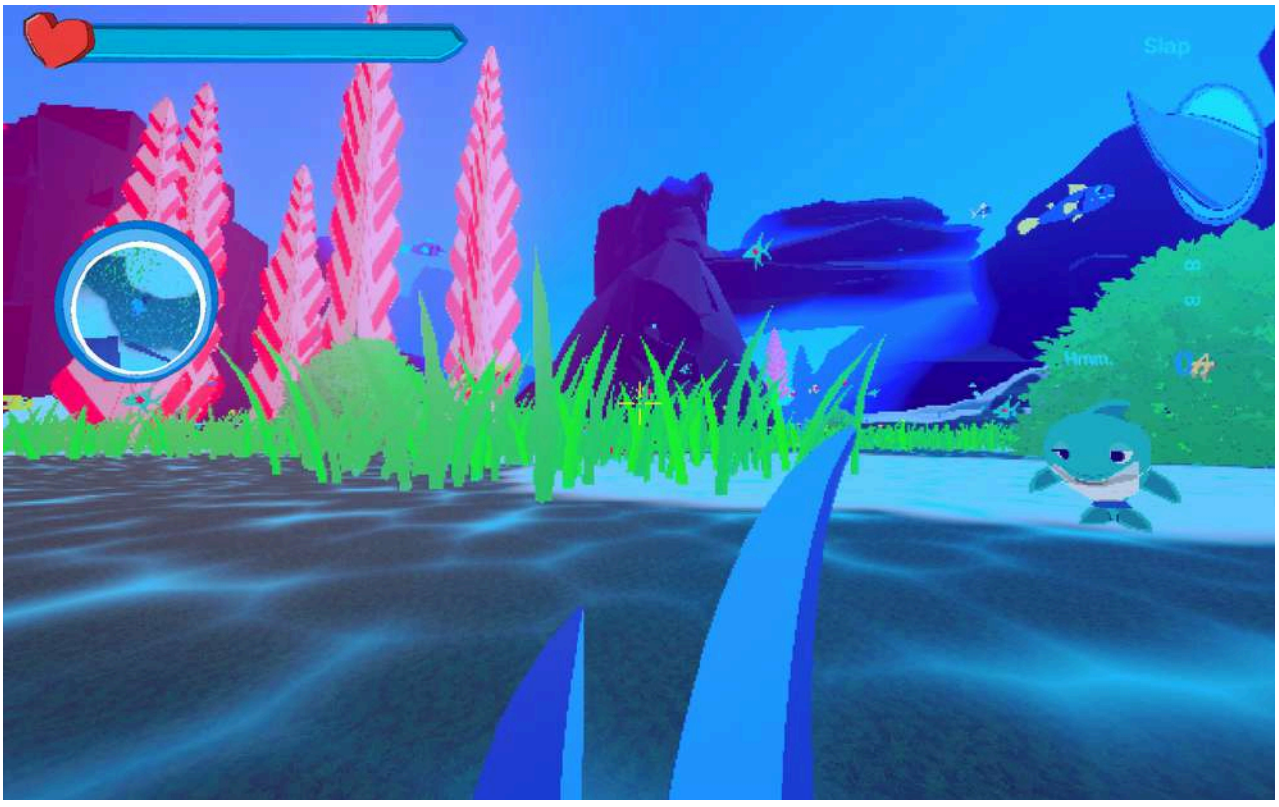
The main techniques and procedures utilized in creating iT AiN'T THAT DEEP are described in this section. It addresses the game's fundamental design choices, user interface layout, navigation organization, coding style, system architecture, and general functionality. Additionally included are any outside tutorials or resources that were used in the development process.

Although the testing chapter contains comprehensive testing procedures, this part highlights significant modifications brought about by testing and explains them in their pertinent domains. All things considered, this chapter offers a complete synopsis of the game's conception, development, and improvement.

DESIGN

Environment

The environment resembles a retro PlayStation 1 to 2 style, increasing the simplicity of the game. The Environment assets(UNL by Innerverse Games, Nature Hybrid Pack by Nicrom) were sourced from the Unity asset store and manipulated according to our art style, while the terrain was created in-house. After a bit of back and forth, we decided to create 2D sprites and assets for the game as 2D sprites blend better with the 3d environment, since reducing the render scale helps flatten out the image, so its less distinguishable



Character Design

3 Major types of characters:

- Boss - The boss - Dilly the ‘Kid’ - is the main villain of the storyline, who is named after the infamous outlaw in the old wild west, Billy the Kid (Portrayed in game by a grumpy orca)
- Goons - Dilly’s goons, dolphins with anchors, and guns (can be killed)
- NPCs - Dolphins and other fish scattered around the environment (can’t be killed)

Dilly Goon NPC Filler NPC

Each character has at least 3 states, with the boss coming up with a total of 24 states. Therefore, 24 different sprites were created to animate the boss character. Similarly, the goons have 13 sprites with all their necessary animations.

3D Design Issues

We started off with a 3D design plan and created multiple models, yet a proper animation system wasn’t possible at such a scale, therefore, that idea was scrapped, and we went a bit old-school with the design and functionality. With the retro pixelated art style, the 2D sprites blend so well with the 3D environment. This level of seamlessness was achieved through this design decision.

3D Buildings

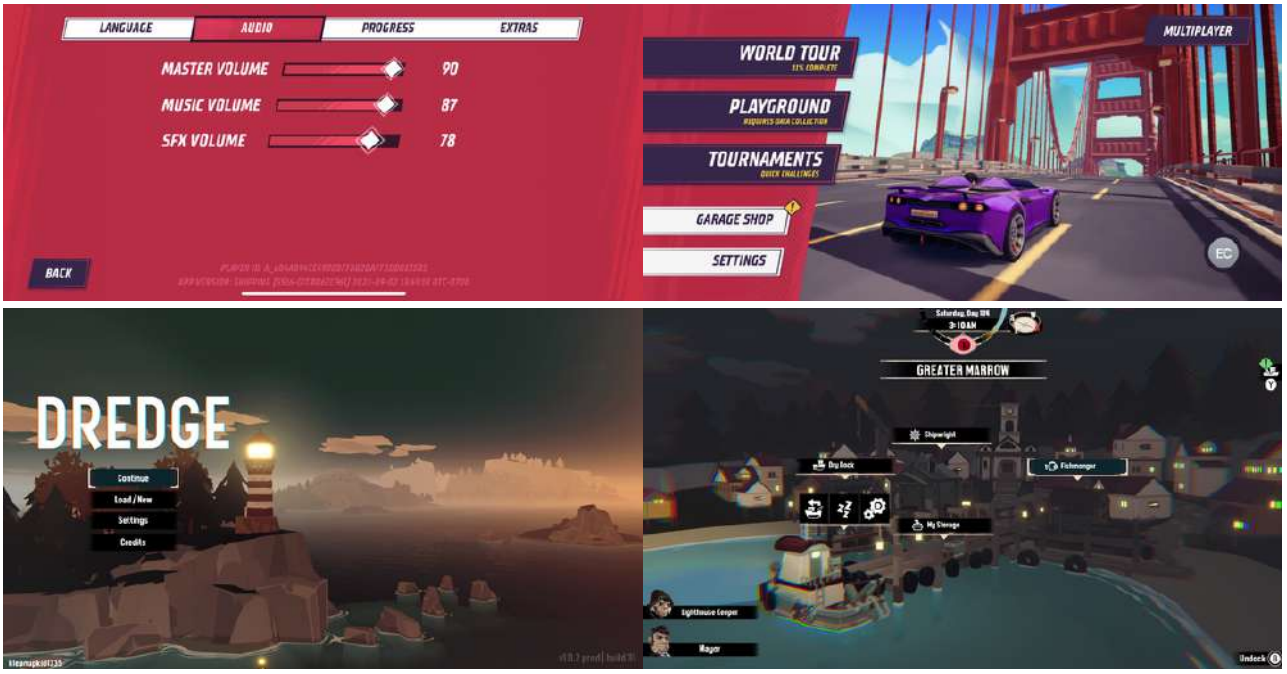
The buildings were created in Blender 3D. We made them in such a way that they are super low-poly and they don’t weigh heavily on the game engine. Therefore, all the buildings were created under 2000 polygons, and texture painted in Blender itself to reduce the dependency on third-party textures.

UI Design

The UI is inspired by the GTA franchise, since the game follows a similar line of gameplay and storyline, it seemed fit to take inspiration from it. It has touches of ‘Hungry Shark’ and Saint’s Row III too. The components are cartoonish and symmetry is absent in most cases.

UI references

I created a vector landscape, and all the UI components have sharp and clunky edges. This was inspired by the game Horizon Chase 2 and Dredge.



INTERACTION FRAMEWORK & USER FLOW

The user flow for this UI is pretty straightforward and intuitive. The UI flows along with the user's needs and makes everything accessible.

Flow 1: Resume Game

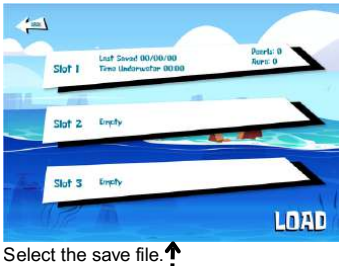
Title screen pops up with a message that says 'press any key'. Clicking anywhere on the screen takes you directly to the Main Menu

Pressing 'play' takes the user to the Play Menu

Press 'Resume'.

Game Loading Screen.

Flow 2: Load Game



Select the save file.

Desired save file selected.

Warning pop up.

Press 'Load Game'.

Pressing 'play' takes the user to the Play Menu

Title screen pops up with a message that says 'press any key'. Clicking anywhere on the screen takes you directly to the Main Menu

Flow 2: Access ‘Options’ while in play

Title screen pops up with a message that says ‘press any key‘. Clicking anywhere on the screen takes you directly to the Main Menu

Pressing ‘play’ takes the user to the Play Menu

Resume or load or new game

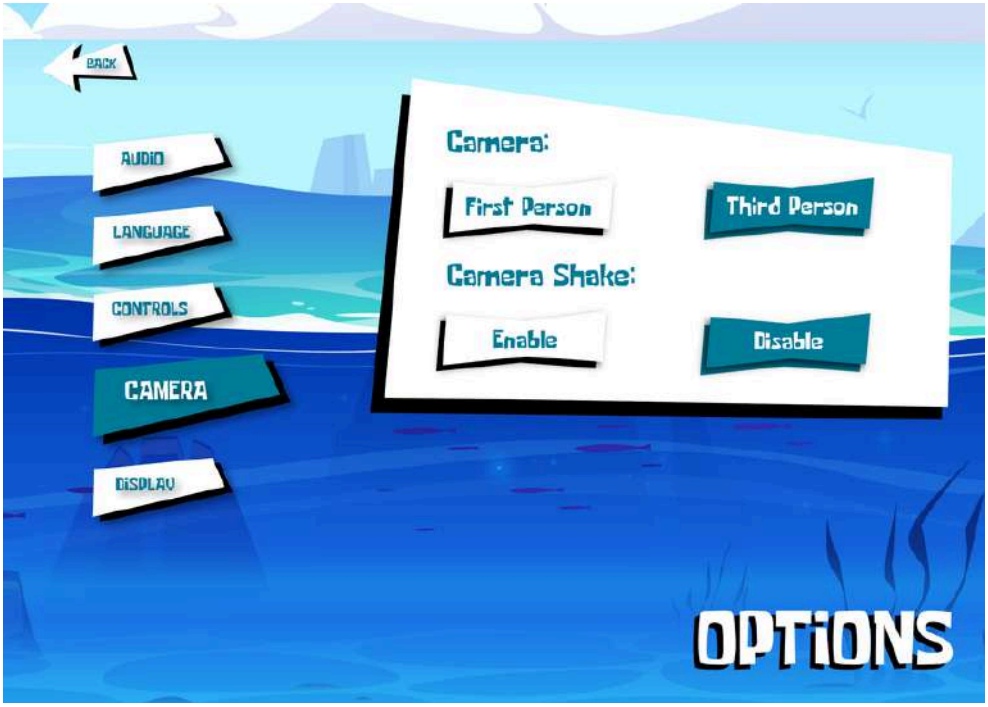
Loading screen

Game screen

Pause Menu

When the options screen comes up the first section is Audio, navigate to camera

Click on camera shake diable



UI/UX & PSYCHOLOGICAL PRINCIPLES

The game integrates predetermined UI/UX and psychological principles to enhance gameplay, hold the attention of the players, and make it more user-friendly.

The major principles in this design are:

Cognitive Load Theory

'Cognitive Load Theory (CLT) - coined in 1988 by John Sweller, suggests that our working memory is only able to hold a small amount of information at any one time and that instructional methods should avoid overloading it in order to maximise learning' (Sweller, 1988).

The simplified UI of "iT AiN'T THAT DEEP" minimizes the distraction that may be caused to the players and will hold their attention longer, which will in turn make the experience more interactive and interesting

Clean and simple options menu, a clear HUD overlay for a better gameplay experience and straightforward message pop-up according to the cognitive load theory

Flow Theory

'Flow theory proposed by Csikszentmihalyi (1990) indicates that when a person becomes engaged in an activity, detrimental thoughts and feelings are pushed aside'. (Csikszentmihalyi, 1990)

The UI is in a constant flow, enabling players to engage in the process without interruptions.

An example of this theory in practice with the starting screens

Visual Hierarchy

'A clear visual hierarchy guides the eye to the most important elements on the page. It can be created through variations in color and contrast, scale, and grouping.' (Nielsen Norman Group, 2021)

This was achieved through clear contrast and spacing, and scale. Important elements and components stand out to the user.

Clear visual hierarchy in play, with contrasting colours and selected elements scaled.

Color Psychology

Color psychology in design is the study of how different colors can influence human emotions, behaviors, and perceptions when applied to various design elements, whether in graphics, interior design, branding, marketing, or any other creative discipline. This field of study is rooted in the idea that colors have the power to evoke specific feelings and convey particular messages, impacting how people perceive and interact with design. (Medium, 2023)

The entire UI features a cool and desaturated color scheme. It has a calming effect to users and the blue stands out as it is set in marine environment

Cool colour scheme on the title page, Colour scheme with hex codes

GAME UI/UX BEST PRACTICES

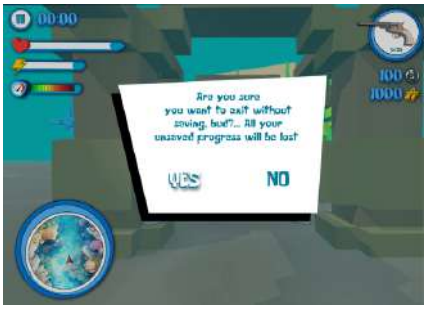
Some key decisions were made according to guidelines provided by the industry expert sources

Unity UI/UX Guidelines

From Unity's UI/UX best practices for games:

- "Show only the information the player needs right now."
- → In-game HUD is minimal; contextual info like sonar or item count only appears when relevant.
- "Reduce clicks and streamline navigation."
- → Key actions like starting the game or adjusting settings are one click from the main menu.

(Unity Learn – UI/UX Best Practices)

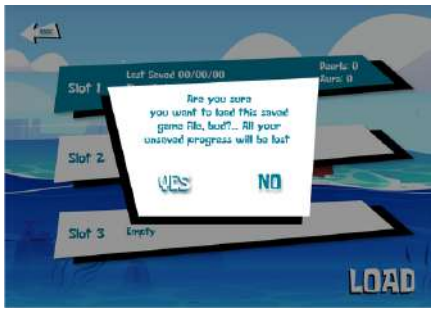


UX of Games Framework

Clarity, Feedback, and Contextuality are three of the ten pillars of game UX (Boudreau, 2017):

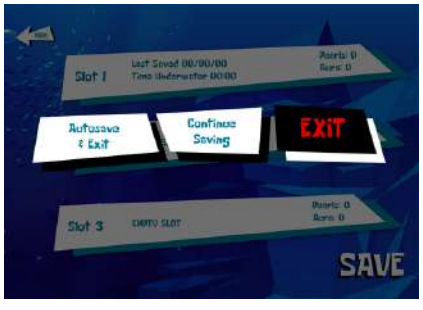
- Clarity: Simple, intuitive iconography and language
- Feedback: Subtle audio/visual cues on hover, click, and collect
- Contextuality: HUD only appears in active gameplay, not menus

(The UX of Games – Celia Hodent)



THE "BIG RED BUTTON"

The big red button has been implemented for the exit option in this game, this was inspired by an UID lecture.



CODE

This section documents the core technical foundations of It Ain't That Deep, outlining how each gameplay, UI, AI, and feedback system was engineered to support the project's philosophy of "structured chaos." It explains the architecture, data flow, and interactions between systems, focusing on clarity, modularity, and extensibility. Together, these components form the underlying framework that enables the game's responsive combat, expressive juicing, and overall player experience.

Weapon data model

```
public class WeaponData : ScriptableObject
{
    42 references
    public enum WeaponType
    {
        Slap,
        Melee,
        Pistol,
        Rifle,
        RPG
    }

    21 references
    public enum DamageType
    {
        Slap,
        Melee,
        Pistol,
        Rifle,
        Explosive
    }
```

Additional fields control hit reaction (knockback and hit-stun), RPG splash radius, ammo sizes and SFX:

```
[Header("Hit Reaction")]
[Tooltip("Force applied to enemies when hit. Higher = more knockback.")]
public float knockbackForce = 6f;

[Tooltip("How long enemies are stunned after being hit.")]
public float hitStunDuration = 0.15f;

[Header("RPG Settings (only used for WeaponType.RPG)")]
[Tooltip("Radius of the sphere cast for RPG splash damage")]
public float sphereCastRadius = 5f;

[Header("Distance and Rate/Reload Settings")]
public float range = 5f;
public float fireRate = 1f;
public float reloadTime = 2f;

[Header("Ammo Settings")]
public int magazineSize;
public int clipSize;

[Header("Audio Settings")]
[Tooltip("Play this when the player equips this weapon")]
public AudioClip switchSFX;

[Tooltip("Play this when the player reloads this weapon")]
public AudioClip reloadSFX;

[Tooltip("Play this when the clip is empty")]
public AudioClip emptySFX;
```

Why:

- Can quickly tune feel (DPS, reload time, RPG splash) without opening the shooter code.
- Separating WeaponType and DamageType allows cosmetic variations (e.g. two rifles with different damage types) while still sharing behaviour.

Weapon inventory & swapping (WeaponManager)

WeaponManager is the central authority for which weapon is equipped and exposes this to the rest of the game:

```
Unity Script (1 asset reference) | 28 references
public class WeaponManager : MonoBehaviour
{
    [Header("Available Weapons")]
    [SerializeField] private List<WeaponData> weaponInventory = new List<WeaponData>();
    // I track the currently equipped weapon for all other systems
    21 references
    public static WeaponData CurrentWeapon { get; private set; }

    // I notify subscribers when the player switches weapons
    public static event Action<WeaponData> OnWeaponSwitched;
    4 references
    public List<WeaponData> GetAllWeapons() => weaponInventory;
```

Input handling uses number keys + mouse-wheel, but is gated by the pause flag:

```
Unity Message | 0 references
private void Update()
{
    if (PauseMenuController.IsPaused)
        return;

    // Number key quick-select: 1 for first, 2 for second, etc.
    for (int index = 0; index < weaponInventory.Count; index++)
    {
        if (Input.GetKeyDown(KeyCode.Alpha1 + index))
        {
            EquipWeaponAtIndex(index);
            return; // I process only one equip per frame
        }
    }

    // Mouse wheel cycling: scroll up = next, scroll down = previous
    float scrollDelta = Input.GetAxis("Mouse ScrollWheel");
    if (scrollDelta > 0f)
    {
        EquipWeaponAtIndex((equippedWeaponIndex + 1) % weaponInventory.Count);
    }
    else if (scrollDelta < 0f)
    {
        int previousIndex = (equippedWeaponIndex - 1 + weaponInventory.Count) % weaponInventory.Count;
        EquipWeaponAtIndex(previousIndex);
    }
}
```

Equipping a weapon updates the static CurrentWeapon, plays its switch SFX and notifies subscribers:

```
    // I equip the weapon at the given inventory index, play sound, and broadcast
    4 references
    private void EquipWeaponAtIndex(int index)
    {
        if (index < 0 || index >= weaponInventory.Count)
            return; // I ignore invalid indices

        equippedWeaponIndex = index;
        CurrentWeapon = weaponInventory[index];
        Debug.Log($"Equipped weapon: {CurrentWeapon.weaponName}");

        // I play the switch sound if assigned
        if (switchAudioSource != null && CurrentWeapon.switchSFX != null)
            switchAudioSource.PlayOneShot(CurrentWeapon.switchSFX);

        // I notify any systems watching for weapon changes
        OnWeaponSwitched?.Invoke(CurrentWeapon);
    }
```

Shooting, ammo and damage

Shooting, ammo and damage (FirstPersonShooter + AmmoDrop)
FirstPersonShooter tracks ammo per WeaponData in a dictionary so each gun remembers its own magazine/reserve when you switch away and back:

```
// I define what counts as a ranged weapon in one central place
8 references
public static bool IsRangedWeapon(WeaponData.WeaponType type) =>
    type == WeaponData.WeaponType.Pistol ||
    type == WeaponData.WeaponType.Rifle ||
    type == WeaponData.WeaponType.RPG;

// I expose ammo and reload state for UI
2 references
public int GetMagazineAmmo() => magazineAmmoCount;
2 references
public int GetReserveAmmo() => reserveAmmoCount;
1 reference
public bool GetIsReloading() => isReloading;
```

When the weapon is switched, ammo state is looked up or initialised from WeaponData:

```
// I reset ammo and state when the player switches weapons
1 reference
private void HandleWeaponSwitched(WeaponData newWeapon)
{
    StopAllCoroutines();
    isReloading = false;

    if (IsRangedWeapon(newWeapon.weaponType))
    {
        if (!ammoStatesByWeapon.TryGetValue(newWeapon, out AmmoState savedState))
        {
            savedState = new AmmoState
            {
                magazine = newWeapon.magazineSize,
                reserve = newWeapon.clipSize
            };
            ammoStatesByWeapon[newWeapon] = savedState;
        }
        magazineAmmoCount = savedState.magazine;
        reserveAmmoCount = savedState.reserve;
    }
    else
    {
        magazineAmmoCount = 0;
        reserveAmmoCount = 0;
    }

    nextFireTimestamp = 0f;
}
```

Ammo pickups (AmmoDrop) add to the reserve of all ranged weapons, using the same helper:

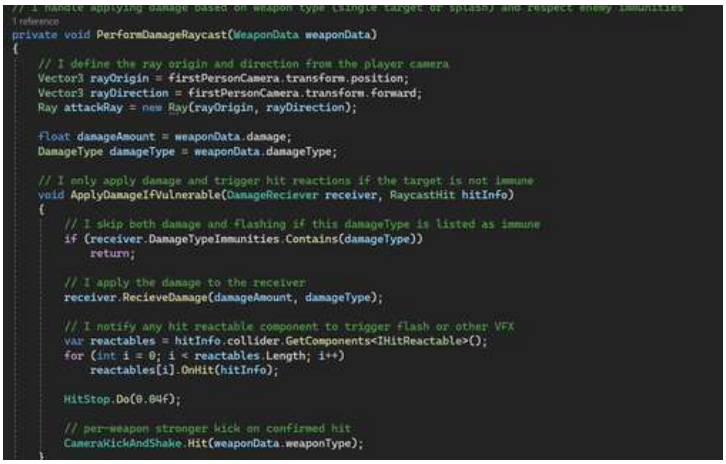
```
1 reference
private void ApplyAmmo(FirstPersonShooter shooter)
{
    var wm = FindFirstObjectByType<WeaponManager>();
    foreach (var weapon in wm.GetAllWeapons())
    {
        if (!FirstPersonShooter.IsRangedWeapon(weapon.weaponType))
            continue;

        int amount = GetAmountForWeapon(weapon.weaponType);
        if (amount > 0)
            shooter.AddReserveAmmoToWeapon(weapon, amount);
    }
}
```

Why:
This keeps ammo logic entirely in the shooter, while pickups just say “give +X rifle ammo”.

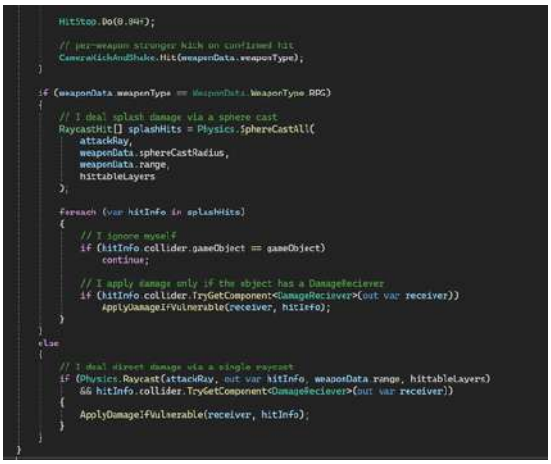
Because the system is keyed by WeaponData, adding a new gun automatically benefits from ammo pickups if it’s included in WeaponManager and IsRangedWeapon.

Damage application and enemy immunities

PerformDamageRaycast is the core routine that calculates hits, applies damage and triggers hit reaction:



```
// I handle applying damage based on weapon type (single target or splash) and respect enemy immunities
1 reference
private void PerformDamageRaycast(WeaponData weaponData)
{
    // I define the ray origin and direction from the player camera
    Vector3 rayOrigin = firstPersonCamera.transform.position;
    Vector3 rayDirection = firstPersonCamera.transform.forward;
    Ray attackRay = new Ray(rayOrigin, rayDirection);

    float damageAmount = weaponData.damage;
    DamageType damageType = weaponData.damageType;

    // I only apply damage and trigger hit reactions if the target is not immune
    void ApplyDamageIfVulnerable(DamageReciever receiver, RaycastHit hitInfo)
    {
        // I skip both damage and flashing if this damageType is listed as immune
        if (receiver.DamageTypeImmunities.Contains(damageType))
            return;

        // I apply the damage to the receiver
        receiver.RecieveDamage(damageAmount, damageType);

        // I notify any hit reactable component to trigger flash or other VFX
        var reactables = hitInfo.collider.GetComponents<IHitReactable>();
        for (int i = 0; i < reactables.Length; i++)
            reactables[i].OnHit(hitInfo);

        HitStop.Do(0.04f);

        // per-weapon stronger kick on confirmed hit
        CameraKickAndShake.Hit(weaponData.weaponType);
    }
```



```
    HitStop.Do(0.04f);

    // per-weapon stronger kick on confirmed hit
    CameraKickAndShake.Hit(weaponData.weaponType);
}

if (weaponData.weaponType == WeaponData.WeaponType.RPG)
{
    // I deal splash damage via a sphere cast
    RaycastHit[] splashHits = Physics.SphereCastAll(
        attackRay,
        weaponData.sphereCastRadius,
        weaponData.range,
        hittableLayers
    );

    foreach (var hitInfo in splashHits)
    {
        // I ignore myself
        if (hitInfo.collider.gameObject == gameObject)
            continue;

        // I apply damage only if the object has a DamageReciever
        if (hitInfo.collider.TryGetComponent<DamageReciever>(out var receiver))
            ApplyDamageIfVulnerable(receiver, hitInfo);
    }
}
else
{
    // I deal direct damage via a single raycast
    if (Physics.Raycast(attackRay, out var hitInfo, weaponData.range, hittableLayers)
        && hitInfo.collider.TryGetComponent<DamageReciever>(out var receiver))
    {
        ApplyDamageIfVulnerable(receiver, hitInfo);
    }
}
```

Key points:

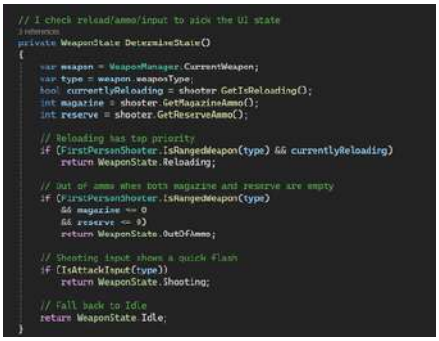
- DamageType-based immunities decide if damage is applied at all.
-
- Interface-based feedback (IHitReactable.OnHit) lets enemies flash, stagger, or spawn particles without the shooter knowing their specifics.
-
- HitStop (tiny 0.04s timescale pause) and camera kick/shake are triggered only on confirmed hits, reinforcing impact; this uses the shared HitStop utility.

Weapon state & HUD integration

Weapon state tracking (WeaponStateTracker)
To drive both the HUD icons and potential future effects, the system tracks a high-level WeaponState:

```
Unity Script (1 asset reference) | 13 references
public class WeaponStateTracker : MonoBehaviour
{
    24 references
    public enum WeaponState
    {
        Idle,
        Shooting,
        Reloading,
        OutOfAmmo
    }

    [Header("Shooting UI Timing")]
    [SerializeField] private float shootingFlashDuration = 0.5f;  // I control how long the shooting icon shows

    4 references
    public static WeaponState CurrentWeaponState { get; private set; } = WeaponState.Idle;
    public static event Action<WeaponState> OnWeaponStateChanged;
```

Each frame, it queries the shooter for ammo and reload status and decides what to show:



```
// I check reload/ammo/input to pick the UI state
1 reference
private WeaponState DetermineState()
{
    var weapon = WeaponManager.CurrentWeapon;
    var type = weapon.weaponType;
    bool currentlyReloading = shooter.GetIsReloading();
    int magazine = shooter.GetMagazineAmmo();
    int reserve = shooter.GetReserveAmmo();

    // Reloading has top priority
    if (FirstPersonShooter.IsRangedWeapon(type) && currentlyReloading)
        return WeaponState.Reloading;

    // Out of ammo when both magazine and reserve are empty
    if (FirstPersonShooter.IsRangedWeapon(type)
        && magazine <= 0
        && reserve <= 0)
        return WeaponState.OutOfAmmo;

    // Shooting input shows a quick flash
    if (IsAttackInput(type))
        return WeaponState.Shooting;

    // Fall back to Idle
    return WeaponState.Idle;
}
```

State-driven icon UI (WeaponStateUIController)

WeaponStateUIController maps each WeaponData to a set of four icons and toggles them based on CurrentWeaponState:

```
Unity Script (1 asset reference) | 0 references
public class WeaponStateUIController : MonoBehaviour
{
    [Serializable]
    4 references
    private struct WeaponStateIcons
    {
        [Header("Icons for different states")]
        public WeaponData weaponData;    // I map this weapon's data to its icons
        public GameObject idleIcon;      // Idle state icon
        public GameObject activeIcon;    // Shooting/attack state icon
        public GameObject reloadingIcon; // Reloading state icon
        public GameObject outOfAmmoIcon; // Out-of-ammo state icon
    }

    [Header("Configure icons for each weapon type below")]
    [SerializeField] private List<WeaponStateIcons> weaponStateIconsList; // I collect inspector configurations

    // I look up icon sets quickly by WeaponData
    private Dictionary<WeaponData, WeaponStateIcons> stateIconsLookup;
```

On enable, it builds a lookup dictionary, subscribes to OnWeaponSwitched and OnWeaponStateChanged, and then:
Hides all icons.
Shows only the icon for the current weapon + current state (with special cases for melee – no reload / out-of-ammo).
This gives very readable visual feedback when reloading, spamming shots, or dry-firing.

```
// I update magazine and reserve ammo labels depending on weapon type
2 references
private void RefreshAmmoDisplay()
{
    if (shooter == null) return;
    var weapon = WeaponManager.CurrentWeapon;
    bool isRanged = weapon.weaponType == WeaponData.WeaponType.Pistol
                 || weapon.weaponType == WeaponData.WeaponType.Rifle
                 || weapon.weaponType == WeaponData.WeaponType.RPG;

    if (isRanged)
    {
        int magCount = shooter.GetMagazineAmmo();
        int reserveCount = shooter.GetReserveAmmo();
        UpdateMagazineLabel(magCount, weapon.magazineSize);
        UpdateReserveLabel(reserveCount);
    }
    else
    {
        // I show "Infinite" for melee weapons
        SetLabelAsInfinite(magazineCountLabel);
        SetLabelAsInfinite(reserveCountLabel);
    }
}
```

Ammo HUD (FirstPersonUIController)
FirstPersonUIController listens to WeaponManager.OnWeaponSwitched and uses the shooter's getters to display ammo:

```
1 reference
private bool WouldDealDamageNow(WeaponData weapon)
{
    if (shooter == null || shooter.FirstPersonCamera == null || weapon == null)
        return false;

    Camera cam = shooter.FirstPersonCamera;
    Vector3 origin = cam.transform.position;
    Vector3 dir = cam.transform.forward;
    float range = weapon.range;
    LayerMask mask = shooter.HittableLayers;

    // Non-RPG: single hitscan ray
    if (weapon.weaponType != WeaponData.WeaponType.RPG)
    {
        if (Physics.Raycast(origin, dir, out _hit, range, mask))
        {
            // Require a DamageReciever and check immunity like PerformDamageRaycast does
            if (_hit.collider.TryGetComponent<DamageReciever>(out var receiver))
            {
                // Skip locking if immune to this weapon's damage type
                return !receiver.DamageTypeImmunities.Contains(weapon.damageType);
            }
        }
        return false;
    }

    // RPG path: mirror  SphereCastAll (splash along the ray)
    Ray attackRay = new Ray(origin, dir);
    RaycastHit[] hits = Physics.SphereCastAll(
        attackRay,
        weapon.sphereCastRadius,
        range,
        mask,
        QueryTriggerInteraction.Ignore
    );

    for (int i = 0; i < hits.Length; i++)
    {
        var h = hits[i];

        // Ignore the player object
        if (shooter != null && h.collider.gameObject == shooter.gameObject)
            continue;

        if (h.collider.TryGetComponent<DamageReciever>(out var receiver))
        {
            if (!receiver.DamageTypeImmunities.Contains(weapon.damageType))
                return true;
        }
    }

    return false;
}
```

Smart crosshair (CrosshairController)
CrosshairController visually connects the weapon data and hit logic to the player's aim:

- It previews hit-validity with a colour: green when the next shot would actually deal damage, white otherwise.

The hit test mirrors the shooter's fire logic and respects damage immunities:

Distribution Strategy

Although the project is developed primarily for Windows, the chosen stack deliberately supports flexible distribution:

- Windows standalone build (.exe) – Primary deliverable for assessment and local playtesting. It offers the best performance and fewest platform constraints.
- WebGL builds – Unity's WebGL support allows exporting the same project to run in a browser. This enables hosting on platforms such as itch.io, which is useful for frictionless external playtesting and sharing the game with peers and supervisors without asking them to install anything.

Overall, the combination of Windows + Unity 6 provides a stable, well-documented, and widely supported technical foundation that lets the project focus on its core contribution: the design and implementation of its systems (weapons, movement, trickshot mechanics, and aura economy), rather than low-level engine or platform issues.

Testing & Feedback-Driven Iteration

The initial external feedback session was conducted on an early prototype of the game. Testers consistently highlighted missing guidance, lack of feedback, unclear goals, difficulty spikes, and an inconsistent art style. Following this, the project underwent substantial iteration. The changes were not superficial; most core systems were expanded or rebuilt entirely. Below is a concise summary of the major improvements introduced *after* the feedback phase.

1. Added NPCs With Contextual Dialogue (Improved Clarity & Guidance)

Early testers noted that the prototype offered **no context or direction**, which made the objective unclear.

To address this, we introduced **interactive NPCs** placed throughout the map, each providing world-building, mechanical hints, or foreshadowing boss encounters.

These NPCs serve three design goals:

- Teach mechanics organically through dialogue.
- Provide narrative context before major encounters (e.g., Dilly the Kid).
- Break up combat pacing and reduce confusion for new players.

This directly addressed feedback about "not knowing what to do next."

2. Difficulty Adjustments for Both Low-Skill and High-Skill Players

Feedback showed a wide gap between players who were familiar with fast FPS games and those who were not.

Post-feedback, we implemented new systems to support *both* audiences:

For less experienced players

- **Collectibles and aura rewards** encourage exploration and give passive bonuses.
- Additional **UI feedback** makes ammo, health, and weapon state far more readable.

For skilled players

- The **Kill Chain system** and **Trickshot system** reward high-performance play.
    - Kill chains escalate tiers, give camera shake, and trigger UI pop-offs.
    - (Implemented in KillChainManager)

Trickshots award bonus aura for advanced movement/spin kills.

(Implemented via TrickshotSpinTracker and TrickshotManager)

These systems allow different playstyles to flourish without compromising the game's pacing.

3. Complete Visual Overhaul to a PS1/PS2 Retro Aesthetic
Testers mentioned the earlier art direction felt inconsistent and generic.
To address this:

- We shifted the entire aesthetic toward a retro PS1/PS2-inspired low-poly look, with stylised textures and simplified shading.
- UI was redesigned to complement this aesthetic, using bolder pixel fonts, heavier outlines, and stronger saturation.
- Added shader adjustments, lower-fidelity texture work, and more stylised post-processing to unify the experience.

This significantly improved visual cohesion and readability.

4. Major Upgrade to Enemy Feedback & Impact
Multiple testers noted the lack of "punch" in hits, including no knockback, no clear feedback when firing, and enemies feeling weightless.
This resulted in a complete overhaul of all hit-feedback systems.
Added post-feedback:

- HitStop micro-pauses on every successful shot
- (HitStop.Do(0.04f)) integrated into shooting logic.
- Enemy knockback was added so that enemies no longer "stick" to the player when hit.
- Knockback emerges naturally from damage and hit reaction systems via IHitReactable and the hit feedback pipeline.

Enemy combat now feels noticeably more reactive and readable.

5. Added Collectibles & Exploration Incentives
Testers asked for "more to do" besides killing enemies.
In response, we added:

- **Collectibles** that bob, rotate and emit SFX/VFX when picked up.
- (AmmoDrop uses this format as baseline bobbing behaviour.)

Collectibles now award **aura points** silently or with UI pop-ups, depending on type.
(AuraPointsManager + AuraFloaterUI)

This gives the level more purpose, creates pacing breaks, and rewards exploration.

6. Expanded Music, UI, SFX, and Hidden Easter Eggs
To increase personality and game "feel," several audiovisual improvements were made:

- **New music tracks** added for different play contexts.
- **Button SFX, UI transitions, and menu interactions** added throughout.
- Added hidden **easter eggs** in menus to enhance personality and replayability.

These changes significantly elevated polish, responsiveness, and aesthetic cohesion.

Summary
The post-feedback phase drove major upgrades across **gameplay**, **visuals**, **feedback systems**, **UI**, **NPC guidance**, and **mechanical inclusivity**.
Nearly every system in the project evolved as a result of user input, and the final version reflects a deliberate effort to improve clarity, accessibility, responsiveness, and style.

Dialogue System:

The game implements a fully custom dialogue system designed specifically for the project's needs. An initial off-the-shelf dialogue asset was rejected because it introduced unnecessary complexity, unused features, and heavy editor tooling that did not suit the scope. The project required a system that could:

- Display short conversations with NPCs
- Trigger rewards (aura or ammo)
- Support branching dialogue
- Provide "barks" reacting to player proximity
- Maintain a lightweight code footprint
- Be fully editable inside Unity without external tools

This resulted in a **three-layer modular architecture**:
(1) Data layer (DialogueNodes),
(2) Runtime controller (DolphinDialogue),
(3) Presentation & behaviour layer (barks, billboarding, and reward logic),
with an additional **custom Unity editor** for fast iteration.

1. Data Layer - DialogueNode System

At the lowest level, each NPC's conversation is defined using **DialogueNode** objects. Each node contains:

- The NPC's written line
- An optional emotional state
- A list of **DialogueOptions** (player choices)

This structure is extremely lightweight but supports full branching conversation flow.

```
using UnityEngine;

9 references
public enum DialogueEmotion
{
    Neutral,
    Positive,
    Negative
}

[System.Serializable]
1 reference
public class DialogueNode
{
    [TextArea(2, 4)]
    public string npcLine;

    public DialogueEmotion emotion = DialogueEmotion.Neutral;

    public DialogueOption[] options;
}

[System.Serializable]
3 references
public class DialogueOption
{
    [TextArea(1, 3)]
    public string playerLine;

    public int nextNodeIndex = -1;
    public bool givesAura;
    public bool givesAmmo;
}
```

Dialogue System:

Why this matters

- The **nextNodeIndex** enables arbitrary branching.
- Rewards are integrated at the *option* level (not node level), which simplifies design: any response can end the conversation and grant ammo or aura.
- The emotion field plugs directly into sprite swapping at runtime.

2. Runtime Layer - DolphinDialogue (Core Conversation Logic)

Each NPC has a DolphinDialogue component that manages proximity detection, conversation flow, emotion changes, SFX, and reward distribution.

2.1 Interaction Radius & Triggering

The NPC checks distance to the player every frame:

```
Unity Message | 0 references
private void Update()
{
    if (PauseMenuController.IsPaused) return;
    if (player == null) return;

    float dist = Vector3.Distance(player.position, transform.position);

    // If you walk away mid-conversation, force end it
    if (inConversation && dist > interactRadius)
    {
        ForceEnd();
        return;
    }

    // If NPC is one-shot and you already finished their dialogue, ignore them
    if (conversationCompleted && oneShot)
        return;

    // Too far to talk
    if (dist > interactRadius) return;

    // Start conversation
    if (Input.GetKeyDown(interactKey) && !inConversation)
        BeginConversation();
}
```

This ensures:

- NPCs only react when the player is close
- Conversation *automatically ends* if the player walks away
- This supports combat scenarios (player can cancel instantly if attacked)

2.2 Conversation Flow

The system keeps track of the current node index:

```
1 reference
private void BeginConversation()
{
    if (conversation == null || conversation.Length == 0)
        return;

    inConversation = true;
    currentNode = 0;
    ShowNode();
}
```

Showing a node pushes its data into the Dialogue UI:

```
2 references
public void ShowNode()
{
    var node = conversation[currentNode];

    SetEmotionSprite(node.emotion);
    PlayTalkSFX();

    // Push the node to the dialogue UI controller
    DialogueUIController.Instance.ShowNode(
        this,
        name,
        node.npcLine,
        node.options,
        node.emotion
    );
}
```

Selecting an option either continues or ends the conversation

```
1 reference
public void ChooseOption(DialogueOption option)
{
    // nextNodeIndex < 0 means "end the conversation"
    if (option.nextNodeIndex < 0)
    {
        // If this option is marked to give Aura, give the configured amount
        if (option.givesAura)
            AuraPointsManager.AddBonusPoints(quipAuraAmount, transform.position, "QUIP");

        // Ammo reward logic
        if (option.givesAmmo)
            GiveAmmoToAllWeapons(5);

        // Mark conversation as completed for one-shot NPCs
        conversationCompleted = true;

        EndConversation();
        return;
    }

    // Otherwise, continue to the next node
    currentNode = option.nextNodeIndex;
    ShowNode();
}
```

Key Features

- **Ends cleanly** when a branch finishes
- **Supports one-shot NPCs** (only talk once)
- **Automatically resets emotion and UI**

Emotional Sprites & SFX

NPCs change facial expression depending on the emotional tone of the line:

```
1 reference
private void SetEmotionSprite(DialogueEmotion emotion)
{
    if (spriteRenderer == null) return;

    switch (emotion)
    {
        case DialogueEmotion.Positive:
            spriteRenderer.sprite = positiveSprite ? positiveSprite : defaultSprite;
            break;
        case DialogueEmotion.Negative:
            spriteRenderer.sprite = negativeSprite ? negativeSprite : defaultSprite;
            break;
        default:
            spriteRenderer.sprite = defaultSprite;
            break;
    }
}
```

A random “talk” sound is played each time the NPC speaks:

```
1 reference
private void PlayTalkSFX()
{
    if (audioSource == null) return;
    if (talkClips == null || talkClips.Length == 0) return;

    var clip = talkClips[Random.Range(0, talkClips.Length)];
    if (clip != null)
        audioSource.PlayOneShot(clip);
}
```

This reinforces the absurd/comedic tone of the game.

Reward Integration

NPC dialogue can give the player aura or ammo via built-in hooks.
Aura:

```
// If this option is marked to give Aura, give the configured amount
if (option.givesAura)
    AuraPointsManager.AddBonusPoints(quipAuraAmount, transform.position, "QUIP");
```

Ammo:

Iterates through all ranged weapons and adds reserve ammo:

```
// Add reserve ammo to every ranged weapon
for (int i = 0; i < weapons.Count; i++)
{
    var weapon = weapons[i];
    if (!FirstPersonShooter.IsRangedWeapon(weapon.weaponType))
        continue;

    shooter.AddReserveAmmoToWeapon(weapon, amount);
}
```

Presentation Layer - NPC Barks & Billboard System

Barks & Attention Radius

NPCs emit short “bark” messages when the player enters a configurable radius:

```
Vector3 pos = transform.position;
pos.y += barkTextYOffset;
barkTMP.transform.position = pos;

barkTMP.transform.rotation = Quaternion.LookRotation(Camera.main.transform.forward);

barkTimer -= Time.deltaTime;
if (barkTimer <= 0f)
{
    barkTimer = barkCooldown;

    if (barkClips != null && barkClips.Length > 0)
        audioSource.PlayOneShot(barkClips[Random.Range(0, barkClips.Length)]);

    if (barkLines != null && barkLines.Length > 0)
    {
        barkTMP.text = barkLines[Random.Range(0, barkLines.Length)];
        barkTMP.gameObject.SetActive(true);
    }
}
```

These function as:

- Flavour text
- Hints
- Contextual jokes
- Indicators that the player is in range to interact

Barks automatically turn off when the player starts full dialogue.

Dialogue Editor - Custom Unity Inspector

To accelerate authoring, the system includes a **custom inspector** that displays dialogue nodes visually.

Features include:

- Collapsible “Node 0 / Node 1 / …” panels
- Multi-line fields for NPC lines
- Player option lists with add/remove buttons
- Dropdown for selecting the next node
- Embedded preview of NPC lines
- “Add New Node” button

This completely eliminates the need for any external tools, conversation branches can be built directly inside Unity in a highly readable format.

Design Rationale

The system was intentionally built from scratch to support the project’s core pillars:

- Absurdity & comedy: NPCs have personality through barks, emotional sprites, and random SFX.
- Clarity: NPC dialogue helps guide players through an otherwise chaotic environment.
- Lightweight mechanics: The game only needed branching choices, rewards, and simple interaction, not a 400-feature dialogue engine.

- **Modularity**: DialogueNodes are pure data. NPC behaviours, emotional logic, interaction distances, and rewards are all configurable per character.
- **Extensibility**: Adding an NPC is as simple as:
  - Dropping a DolphinDialogue component,
  - Filling nodes in the custom editor,
  - Adding bark lines,
  - Setting a radius.

The final system matches the structural complexity of the game, simple where needed, expressive where it contributes to tone and comedy.

Enemy AI System (Core Logic Only)

The game uses a modular AI architecture built around **simple, deterministic behaviours** designed to support fast-paced arcade FPS combat.

Each enemy type follows the same structural principles:

1. **Activation radius** – enemies are idle until the player gets close enough.
2. **Orientation** – enemies always rotate to face the player horizontally.
3. **Engagement warmup** – enemies telegraph attacks by briefly entering a warmup state.
4. **Attack cycle** – melee punch, projectile shot, or stationary snipe depending on type.
5. **AI suspension** – enemies stop acting when stunned, when the game is paused, or when the player is in a respawn phase.

All behaviours use lightweight vector math and minimal physics operations to maintain performance even with many enemies alive.

Melee Enemy AI

File: EnemyMeleeAI.cs

This class defines the behaviour for close-range melee attackers who chase the player and punch at short range.

Activation & Basic Requirements

The AI remains idle until the player enters a configurable activation radius:

```
if (!activated)
{
    float distToPlayer = Vector3.Distance(transform.position, player.position);
    if (distToPlayer <= activationRadius)
        activated = true;
    else
        return;
}
```

This ensures enemies do not chase the player from across the map and reduces CPU load.

The script also suspends behaviour when:

- The game is paused (PauseMenuController.IsPaused)
- The player is respawning (BossController.PlayerIsRespawning)
- The enemy is stunned (EnemyBaseAI.IsStunned)

This unifies AI behaviour with global gameplay states.

Movement & Facing

The melee enemy continuously rotates to face the player on the horizontal plane:

```
1 reference
private void Face(Vector3 flatToPlayer)
{
    if (flatToPlayer.sqrMagnitude <= 0.0001f) return;

    Quaternion target = Quaternion.LookRotation(flatToPlayer.normalized);
    transform.rotation = Quaternion.RotateTowards(transform.rotation, target, turnSpeed * Time.deltaTime);
}
```

While outside punch range, it advances using Rigidbody-based movement:

```
if (dist > punchRange)
{
    if (spriteState != null) spriteState.ShowAdvancing();
    warmupUntil = 0f;

    Vector3 step = flat.normalized * moveSpeed * Time.deltaTime;
    if (!rb.isKinematic)
        rb.MovePosition(rb.position + step);

    return;
}
```

What this achieves:

- Predictable and readable melee behaviour
- Clean rotation without physics jitter
- Smooth chasing that respects physics collisions when needed

Attack Logic

Once within punch distance, the enemy enters a warmup period:

```
if (warmupUntil == 0f)
    warmupUntil = Time.time + engageWarmup;
```

After telegraphing, the enemy punches on cooldown:

```
if (Time.time >= nextPunchTime)
{
    TryPunch();

    // play melee swing sound
    GetComponent<EnemyAudioController>()?.PlayAttackSound();

    if (spriteState != null) spriteState.PulseAttackFollowThrough();
    nextPunchTime = Time.time + punchCooldown;
}

else
{
    if (spriteState != null) spriteState.ShowEngaging();
}
```

Damage is applied directly to the player's DamageReciever:

```
1 reference
private void TryPunch()
{
    if (player == null) return;

    if (player.TryGetComponent(out DamageReciever dr))
    {
        dr.RecieveDamage(punchDamage, WeaponData.DamageType.Melee);
        return;
    }

    var drParent = player.GetComponentInParent<DamageReciever>();
    if (drParent != null)
        drParent.RecieveDamage(punchDamage, WeaponData.DamageType.Melee);
}
```

Mobile Sniper AI

File: EnemySniperAI.cs
A long-range unit that moves until in firing position, waits through a warmup, then fires projectiles.

```
if (dist > attackRange)
{
    if (spriteState != null) spriteState.ShowAdvancing();
    warmupUntil = 0f;

    Vector3 step = flat.normalized * moveSpeed * Time.deltaTime;
    if (rb != null && !rb.isKinematic)
        rb.MovePosition(rb.position + step);
    else
        transform.position += step;

    return;
}
```

Activation
Identical activation pattern to melee enemies.

Movement Toward Attack Range
- If outside attackRange, the sniper advances:

Within range, movement stops and firing begins.

Aiming & Attack Cycle
The sniper faces the player, warms up, then fires:

```
if (warmupUntil == 0f)
    warmupUntil = Time.time + engageWarmup;

if (Time.time < warmupUntil)
{
    if (spriteState != null) spriteState.ShowEngaging();
    return;
}

if (Time.time >= nextFireTime)
{
    TryShoot();
    if (spriteState != null) spriteState.PulseAttackFollowThrough();
    nextFireTime = Time.time + fireCooldown;
}
else
{
    if (spriteState != null) spriteState.ShowEngaging();
}
```

Projectile instantiation:

```
1 reference
private void TryShoot()
{
    if (projectilePrefab == null || muzzle == null || player == null) return;

    Vector3 targetPoint = player.position;
    Vector3 dir = (targetPoint - muzzle.position).normalized;

    EnemyProjectile proj = Instantiate(projectilePrefab, muzzle.position, Quaternion.LookRotation(dir));
    proj.Init(transform, player.position);

    // play gunshot attack sound
    GetComponent<EnemyAudioController>()?.PlayAttackSound();
}
```

What this achieves:
- Aim behaviour is consistent and always horizontal.
- Warmup provides fairness, snipers never instantly fire when entering LOS.
- Projectile logic ties into the wider damage / hit reaction ecosystem.

Stationary Sniper AI

File: StationarySniperAI.cs

A turret-style variant that does not move, but otherwise follows the same behaviour model.

Differences from mobile sniper:

- No movement logic (only rotates to face the player).
- Larger attack range (up to 120m).
- Uses a simplified sprite controller (StationarySniperSpriteController).
- Aggro behaviour still uses activation radius to avoid processing out-of-range enemies.

Core firing logic and warmup cycle remain identical:

```
if (dist > attackRange)
{
    if (spriteState != null) spriteState.ShowEngaging();
    warmupUntil = 0f;
    return;
}

if (warmupUntil == 0f)
    warmupUntil = Time.time + engageWarmup;

if (Time.time < warmupUntil)
{
    if (spriteState != null) spriteState.ShowEngaging();
    return;
}

if (Time.time >= nextFireTime)
{
    TryShoot();
    if (spriteState != null) spriteState.PulseAttackFollowThrough();
    nextFireTime = Time.time + fireCooldown;
}
else
{
    if (spriteState != null) spriteState.ShowEngaging();
}
```

What this achieves:

- Stationary snipers act as environmental hazards and lane control units.
- Simple but effective AI loop with high legibility.

Enemy Sprite Controllers

Enemies call:

- ShowEngaging() when tracking or warming up
- PulseAttackFollowThrough() when firing

These calls *do not* change gameplay, but they ensure the AI's internal state is reflected visually.

Full explanation of the pulsing and low-HP variants will be included in the juicing section.

Enemy Spawning System (Core AI Infrastructure)

File: EnemySpawner.cs

This system manages enemy population and ensures combat pacing remains controlled.

Spawn Budget & Concurrency Limits

```
Unity Script (2 asset references) | 0 references
public class EnemySpawner : MonoBehaviour
{
    [Header("Spawn Settings")]
    [SerializeField] private GameObject[] enemyPrefabs;

    [Tooltip("Maximum enemies alive at once.")]
    [SerializeField] private int maxConcurrentEnemies = 5;

    [Tooltip("Total enemies allowed to be spawned over time.")]
    [SerializeField] private int totalSpawnBudget = 20;

    [SerializeField] private float spawnDelay = 1f;
    [SerializeField] private Vector2 spawnAreaSize = new Vector2(20f, 20f);
    [SerializeField] private float minSpawnDistance = 2f;
    [SerializeField] private float spawnHeight = 0f;
```

This ensures:

- The map is never overwhelmed
- Spawns scale throughout gameplay
- CPU load remains stable

Spawner Loop

The spawner continually attempts to spawn until the total budget is exhausted:

```
1 reference
private IEnumerator SpawnerLoop()
{
    spawning = true;

    while (totalSpawned < totalSpawnBudget)
    {
        // Wait until we are below the max allowed enemies
        while (activeEnemies.Count >= maxConcurrentEnemies)
        {
            yield return null;
        }

        // Attempt a spawn
        TrySpawnOne();

        yield return new WaitForSeconds(spawnDelay);
    }

    spawning = false;
}
```

Position Selection

Random positions within a square area around the spawner:

```
for (int i = 0; i < maxAttempts; i++)
{
    candidate = GetRandomPosition();
    if (IsPositionValid(candidate))
    {
        valid = true;
        break;
    }
}
```

minSpawnDistance ensures spacing between spawns:

```
1 reference
private bool IsPositionValid(Vector3 pos)
{
    for (int i = 0; i < usedPositions.Count; i++)
    {
        if (Vector3.Distance(pos, usedPositions[i]) < minSpawnDistance)
            return false;
    }
    return true;
}
```

Death Tracking

Enemies are removed from activeEnemies when their DamageReciever fires an onDeath event:

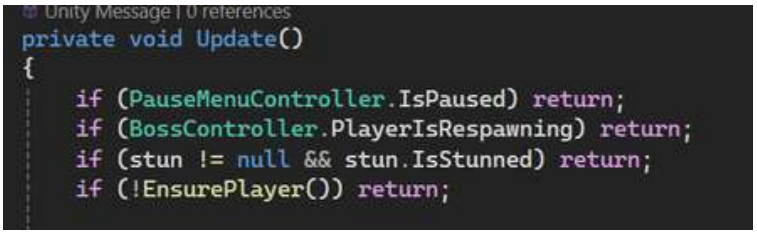
```
// Track when enemy dies
var dr = enemy.GetComponent<DamageReciever>();
if (dr != null)
{
    dr.onDeath.AddListener(() =>
    {
        activeEnemies.Remove(enemy);
    });
}
else
{
    // fallback: auto-remove after destroy
    StartCoroutine(RemoveOnDestroy(enemy));
}
```

How this links to the rest of the AI:

- Snipers and melee enemies all use the same DamageReciever component.
- Spawners therefore don't need to know specifics- death is unified and generic.

Shared AI Architecture (Design Summary)
Across all enemy types, the AI shares several common patterns:
Centralised player reference resolution
All enemies self-resolve the player target:

Ensures enemies continue functioning even if player objects change or respawn.
Unified stun / pause gating
All enemies check:



This blocks all AI behaviour globally under certain states, ensuring systemic consistency.
Consistent warmup → attack loop
All enemies support:

- Telegraph phase
- Attack cooldown
- Attack execution event
- State updates to sprite controller

This makes enemy behaviour predictable, fair, and readable to the player.
Conclusion
The enemy AI system is deliberately simple but extremely consistent, supporting a fast-paced arcade FPS loop.
Each enemy type follows the same activation/movement/engagement pattern with minimal branching logic.
AI interacts cleanly with:

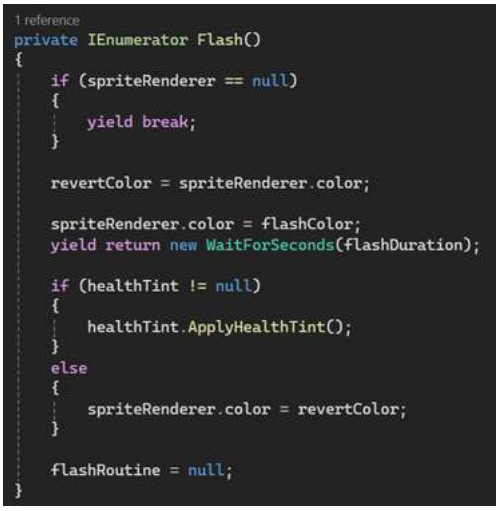
- DamageReciever for damage
- Spawners for lifecycle management
- Pause/Respawn systems
- Sprite controllers (visual feedback)
- Global gameplay states

This modularity is what allowed the game to scale from early prototypes to full combat arenas without rewriting core logic.

Hit Flash (Immediate Damage Feedback)

EnemyFlash.cs

When an enemy is hit, they briefly flash a high-contrast color:



```
1 reference
private IEnumerator Flash()
{
    if (spriteRenderer == null)
    {
        yield break;
    }

    revertColor = spriteRenderer.color;

    spriteRenderer.color = flashColor;
    yield return new WaitForSeconds(flashDuration);

    if (healthTint != null)
    {
        healthTint.ApplyHealthTint();
    }
    else
    {
        spriteRenderer.color = revertColor;
    }

    flashRoutine = null;
}
```

What it achieves:

- Instant recognition of successful hits.
- Works for all weapons (“universal feedback pipeline”).
- Respects low-health tint after flashing by re-applying EnemyHealthTint.

Links to:

- DamageReciever.onHit triggers this component.
- IHitReactable ensures the shooter registers hit feedback only when damage applies.

Enemy Juicing Systems

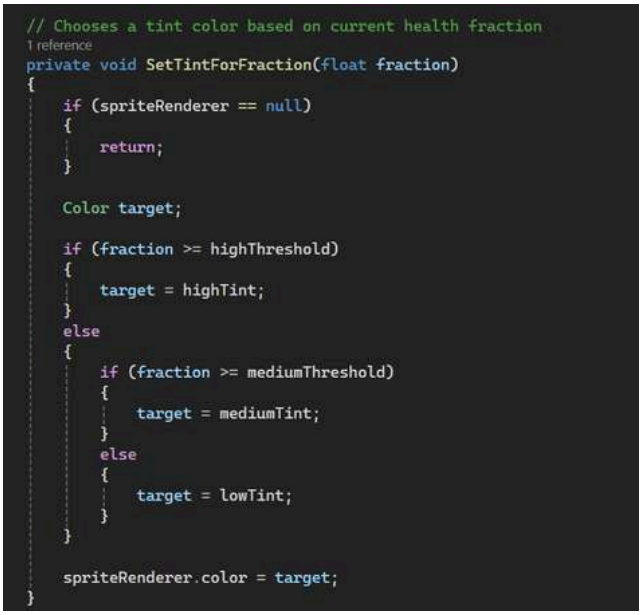
To reinforce the game’s core pillar of exaggerated, arcade-style feedback, every enemy integrates a suite of “juice” systems. These systems do **not** affect gameplay logic; instead, they communicate impact, danger, and personality. The juicing layer is fully modular and plugs into the core AI via DamageReciever and IHitReactable.

The following covers the major components.

Health Tinting (Persistent Damage State)

EnemyHealthTint.cs

Enemies gradually shift colour based on remaining health:



```
// Chooses a tint color based on current health fraction
1 reference
private void SetTintForFraction(float fraction)
{
    if (spriteRenderer == null)
    {
        return;
    }

    Color target;

    if (fraction >= highThreshold)
    {
        target = highTint;
    }
    else
    {
        if (fraction >= mediumThreshold)
        {
            target = mediumTint;
        }
        else
        {
            target = lowTint;
        }
    }

    spriteRenderer.color = target;
}
```

What it achieves:

- A readable health bar without UI clutter.
- Communicates “threat level” (low HP enemies turn red).
- Integrates naturally with flash effects.

Links to:

- DamageReciever.onHit and .onDeath.
- Sprite controllers which pick low-HP animation frames based on GetHealthFraction().

Sprite State Controllers (Animation Feedback)
EnemySpriteStateController.cs / StationarySniperSpriteController.cs
These controllers handle all visual state changes for walking, engaging, attacking, and hit overrides. Example attack pulse:

```
    // play melee swing sound
    GetComponent<EnemyAudioController>()?.PlayAttackSound();

    if (spriteState != null) spriteState.PulseAttackFollowThrough();
    nextPunchTime = Time.time + punchCooldown;
}

else
{
    if (spriteState != null) spriteState.ShowEngaging();
}
```

What it achieves:

- Attack frames with anticipation and follow-through.
- Hit frames that override movement and engagement frames.
- Low-health variants for all animations.
- Flipbook walking to mimic PS1-style sprite animation.

Links to:

- AI calls ShowAdvancing(), ShowEngaging(), and PulseAttackFollowThrough() depending on behaviour.
- DamageReciever.onHit triggers hit pulses.

```
[RequireComponent(typeof(SpriteRenderer))]
Unity Script (7 asset references) | 2 references
public class EnemySpriteStateController : MonoBehaviour
{
    7 references
    public enum State { Advancing, Engaging }
    11 references
    private enum PulseType { None, Hit, Attack }

    [Header("References")]
    [SerializeField] private SpriteRenderer spriteRenderer;

    [Header("Advancing (Walk) — Two Frames (Normal)")]
    [SerializeField] private Sprite walkFrameA;
    [SerializeField] private Sprite walkFrameB;
    [SerializeField] private float walkFps = 8f;

    [Header("Advancing (Walk) — Two Frames (Low HP)")]
    [SerializeField] private Sprite lowWalkFrameA;
    [SerializeField] private Sprite lowWalkFrameB;

    [Header("Engaging (Normal)")]
    [SerializeField] private Sprite engagingFrame;

    [Header("Engaging (Low HP)")]
    [SerializeField] private Sprite lowEngagingFrame;

    [Header("Attack / Hit Override Sprites (Normal)")]
    [SerializeField] private Sprite attackFollowThroughFrame;
    [SerializeField] private Sprite hitFrame;

    [Header("Attack / Hit Override Sprites (Low HP)")]
    [SerializeField] private Sprite lowAttackFollowThroughFrame;
    [SerializeField] private Sprite lowHitFrame;

    [Header("Timings")]
    [SerializeField] private float attackPulseSeconds = 0.20f;
    [SerializeField] private float hitPulseSeconds = 0.10f;

    [Header("Low-HP Threshold")]
    [SerializeField] private float lowHealthThreshold = 0.40f;

    private State state = State.Advancing;
    private PulseType pulse = PulseType.None;
```

Knockback & Hit-Stun (Physical Impact Feedback)
KnockbackOnHit.cs
Enemies physically recoil based on current weapon's knockback force:

```
2 references
public void OnHit(RaycastHit hitInfo)
{
    // Do nothing if dead or disabled
    if (dr == null || dr.CurrentHealth <= 0f || !gameObject.activeInHierarchy)
        return;

    var weapon = WeaponManager.CurrentWeapon;
    if (weapon == null) return;

    Vector3 dir = (transform.position - hitInfo.point).normalized;
    dir = (dir + Vector3.up * 0.2f).normalized;

    rb.AddForce(dir * weapon.knockbackForce, ForceMode.VelocityChange);

    if (aiBase != null && weapon.hitStunDuration > 0f)
        StartCoroutine(StunRoutine(weapon.hitStunDuration));
}
```

```
1 reference
private IEnumerator StunRoutine(float duration)
{
    // Abort if disabled mid-stun
    if (isStunned || dr.CurrentHealth <= 0f || !gameObject.activeInHierarchy)
        yield break;

    isStunned = true;
    aiBase.SetStunned(true);

    yield return new WaitForSeconds(duration);

    aiBase.SetStunned(false);
    isStunned = false;

    rb.linearVelocity *= postKnockbackDamping;
}
```

← Includes temporary stun.

What it achieves:

- Makes hits feel punchy and weighty.
- Temporarily disables AI to create windows of aggression for the player.
- Visually reinforces weapon differences.

Links to:

- WeaponData knockback + stun values.
- AI respects EnemyBaseAI.IsStunned before moving or attacking.

Audio Feedback (Hit / Attack / Death Sounds)
EnemyAudioController.cs / DeathAudioPrefab.cs
Hit, attack, and death events play randomised sound variants with pitch variation:

```
2 references
private void PlayHitSound(float dmg, WeaponData.DamageType type)
{
    if (hitSounds == null || hitSounds.Length == 0) return;

    AudioClip clip = hitSounds[Random.Range(0, hitSounds.Length)];

    audioSource.Stop();
    audioSource.pitch = Random.Range(hitPitchRange.x, hitPitchRange.y) * SloMo.CurrentPitchScale;
    audioSource.PlayOneShot(clip, hitVolume);
}

3 references
public void PlayAttackSound()
{
    if (attackSounds == null || attackSounds.Length == 0) return;

    AudioClip clip = attackSounds[Random.Range(0, attackSounds.Length)];

    audioSource.Stop();
    audioSource.pitch = Random.Range(attackPitchRange.x, attackPitchRange.y) * SloMo.CurrentPitchScale;
    audioSource.PlayOneShot(clip, attackVolume);
}
```

Death sounds optionally spawn a **3D audio prefab** so audio continues even after the enemy object is destroyed:

```
2 references
private void PlayDeathSound()
{
    if (deathPlayed) return;
    deathPlayed = true;

    if (deathSounds == null || deathSounds.Length == 0) return;

    AudioClip clip = deathSounds[Random.Range(0, deathSounds.Length)];

    if (deathAudioPrefab != null)
    {
        GameObject obj = Instantiate(deathAudioPrefab, transform.position, Quaternion.identity);

        AudioSource src = obj.GetComponent<AudioSource>();
        src.pitch = Random.Range(deathPitchRange.x, deathPitchRange.y) * SloMo.CurrentPitchScale;
        src.PlayOneShot(clip, deathVolume);

        Destroy(obj, clip.length + 0.5f);
    }
    else
    {
        audioSource.Stop();
        audioSource.pitch = Random.Range(deathPitchRange.x, deathPitchRange.y) * SloMo.CurrentPitchScale;
        audioSource.PlayOneShot(clip, deathVolume);
    }
}
```

What it achieves:

- Ensures consistency of spatial audio even if enemy is immediately removed.
- Adds violent, comedic character to encounters.
- Pitch scaling integrates with the slo-mo system for seamless slow-motion audio.

Links to:
DamageReciever.onHit and .onDeath.

7. Kill Feedback: Aura, Kill Chain, and Healing

- *EnemyKillNotifier.cs*
- On death, enemies inform global scoring systems.

Additional gameplay-juice rewards:

- **Ammo drops** via weighted RNG
- **Player healing** (scales with missing HP)
- **Aura additions** via KillChainManager → AuraPointsManager pipeline
- **Floating text** through AuraFloaterUI

What it achieves:

- Makes kills feel rewarding and impactful.
- Links individual enemy deaths to global systems like combos and trickshots.

8. Billboard Facing & Minimap Marker

EnemyBillboard.cs

Faces sprites toward the player for a retro Doom/Quake effect

Also spawns a minimap marker that mirrors orientation

What it achieves:

- Ensures enemies are readable from all angles.
- Cohesive low-poly retro aesthetic.

Enhances navigation clarity.

Summary

The juicing layer transforms basic AI/damage interactions into satisfying, readable, arcade-style combat.

Key qualities:

- **Visual clarity:** flashes, tints, animation pulses.
- **Physical impact:** knockback, hit-stun, physics impulses.
- **Audio character:** variable pitch hit/attack/death sounds.
- **Reward feedback:** aura, ammo, healing, floating text, kill chains.
- **Cohesive presentation:** billboarded sprites, minimap markers, death VFX.

These systems work *together* to ensure every hit, kill, and encounter feels loud, punchy, rewarding, and unmistakably readable, deeply supporting the game's core pillars of absurdity and hyper-feedback.

Player Juicing Systems

- The player juicing layer is responsible for making movement, combat, and scoring feel expressive, powerful, and readable.
- Unlike core gameplay systems, these components do **not** change any mechanical outcomes-their job is to enhance clarity, responsiveness, and the game's exaggerated arcade aesthetic.
- The juicing system is composed of several independent modules that all hook into existing events (weapon fire, player hit, slow-motion activation, kill events, sprinting, etc.).

Camera Motion Feedback

Head-bob & Strafe Roll

(CameraHeadBobTilt)

The camera applies sinusoidal vertical movement while walking, scaled up during sprinting for added intensity. Lateral movement also applies a small roll rotation. These effects create a sense of physicality and momentum, especially in fast movement sequences.

The system automatically disables while paused and responds directly to WASD input and the global sprint flag.

Camera Kick, Screen Shake & FOV Punch

(CameraKickAndShake)

This is the game's largest individual juicing system. Every weapon shot and weapon hit triggers:

- A kickback rotation on the camera
- Screen shake with Perlin noise, decaying over time
- A temporary FOV increase ("FOV punch")
- Optional hitstop when the player is damaged

Weapon type (pistol, rifle, melee, RPG) determines the strength of each effect, and being hit multiplies these values. Sprinting increases shake amplitude.

All effects are purely visual, aiming is unaffected because the camera's final position/rotation is only modified visually after raycasts occur.

Hitstop (Micro-Pause on Impact)

(HitStop)

When a shot lands (via the weapon system) or the player takes damage, the game briefly sets the global timescale to zero for a few milliseconds.

This provides:

- Strong impact feedback
- Extra time readability during chaotic combat
- A retro arcade punchiness

Hitstop respects an internal lock so it cannot be retriggered during an existing pause and does not interfere with normal pausing.

On-Screen Comic Bursts

(ComicBursts)

Every time the player fires a weapon, a random stylised "comic burst" sprite briefly appears on the HUD. It picks a random sprite, size, position, and rotation, fades out, and deletes itself.

Because these bursts appear only on weapon fire events, they help sell the comedic, over-the-top tone of the game and reinforce each shot's impact.

Sprint VFX
(SprintParticleSystem)
When the player begins sprinting, a particle system attached to their feet triggers a continuous sprint trail. When sprinting stops, the system cleanly shuts off.
This is tied directly to the global sprint flag, requiring no other logic.

Aura Floater Feedback
(AuraFloaterUI)
Whenever the player gains aura, either from kills, trickshot bonuses, or dialogue rewards, the game spawns a floating number at the relevant world position.
The text drifts upward and fades out before being automatically removed.
This system ensures resources feel earned rather than abstract and ties together scoring, trickshots, and exploration feedback.

Slow-Motion System
Global Slow-Mo Controller
(SloMo)
Holding the designated key places the game into slow-motion by reducing the global timescale and physics timestep.
The system also:
- Plays a whoosh sound on entering/exiting
- Broadcasts a global pitch scale event
- Provides a fixed slow-mo FOV that the camera system can read
- Automatically resets itself when disabled

This acts not only as a power fantasy mechanic but also a strategy tool for trickshots and kill chains.

Audio Pitch Sync
(SloMoAudio)
Any sound source that needs to react to time-slowing can subscribe to the global pitch event.
This ensures:
- Enemy audio
- Player SFX
- Environmental sounds

all shift pitch according to the slow-motion multiplier for cinematic consistency.

Trickshot Detection & Rewards
Spin Tracking
(TrickshotSpinTracker)
Tracks how many degrees the player rotates during slow-mo.
If the accumulated rotation passes the threshold (e.g., ~300°), the system flags a “trickshot” state.
This state clears automatically if slow-mo ends.
Bonus Application
(TrickshotManager)
When the player kills an enemy, the trickshot manager checks whether a valid trickshot was active.
If so, it:
- Grants bonus aura
- Spawns a special floater UI message
- Plays a trickshot SFX

This integrates slow-mo, spinning, kill feedback, and scoring into a coherent “style system” without complicating core combat mechanics.

Kill Chain Integration
While not strictly a camera or movement juice system, the kill chain effects contribute strongly to player feedback.
Kills trigger escalating UI shakes, labels (e.g., "RAMPAGE"), sound effects, and aura bonuses.
All of this ties into the player juicing category because kill chains provide direct, moment-to-moment reinforcement for aggressive play.

Summary
The player juicing stack spans camera motion, particle effects, on-screen bursts, hitstop, slow-motion, trickshots, and score floaters.
Together, these components achieve:

- **High readability**: hits, shots, damage, and rewards are visually obvious.
- **High impact**: strong sensory reinforcement across audio, camera, and UI.
- **High style**: slow-mo spins, comic bursts, and FOV punches create a distinctive arcade identity.
- **High modularity**: each subsystem is independent, event-driven, and does not interfere with core gameplay logic.

This approach ensures the game feels expressive and energetic without compromising clarity or mechanical depth.

Player Movement, Survivability & Feedback Systems
This section documents the core player-side systems responsible for movement, survivability, and sensory feedback. These systems collectively define how the player *feels* to control, aligning with the project's pillars of responsiveness, high feedback density, and low friction. The implementations draw from several integrated scripts: **FirstPersonController**, **FirstPersonCamera**, **PlayerSecondLifeController**, **PlayerHealthController**, and **PlayerFootstepAudio**.

First-Person Movement Controller
The **FirstPersonController**
provides a physics-based locomotion model using a Rigidbody, allowing for grounded sprinting, strafing, air-control limitations, and a unique "super jump" mechanic.
Key Behaviours
Movement Input → Velocity Output:

- Horizontal inputs are read each frame and resolved in *FixedUpdate* into two axes of ground-relative motion. Vertical velocity is left untouched so gravity feels natural.

Sprint Multiplier:

- Sprinting is toggled via a key, modifying the move speed by a configurable multiplier. This state also feeds into multiple juicing features elsewhere (camera FOV, VFX trails, audio variations).

Ground Check:

- A sphere-based ground check near the collider's feet ensures robust grounding even on slopes, which avoids jitter and unwanted airborne states.

Super Jump:

- When sprinting and grounded, the player can perform a high-impulse jump. Any downward momentum is cancelled to prevent "muddy" jumps.

Design Rationale
This controller intentionally favours *forgiveness and responsiveness* over realism. It is tuned to support the game's expressive movement style (fast sprinting, height variation, high agility).

First-Person Camera System

The **FirstPersonCamera** script
manages mouse look, field-of-view (FOV) changes, and an optional airborne “free rotate” mode.

Key Behaviours

Standard Mouse Look:

- Clamped vertical rotation prevents flips, while yaw rotates the player body.

Dynamic FOV:

- FOV blends between base and sprint values, and is overridden during slow-motion states. This feeds back motion state visually without needing UI.

Free Look Mode:

- While airborne and holding RMB, the camera decouples from the player body, allowing spins, rolls, and advanced trickshot behaviour. Upon grounding, the camera re-aligns with the player.

Design Rationale

The camera system is central to the game’s spectacle: extreme FOV changes, trick-rotate freedom, and smooth transitions all reinforce the high-juiciness and arcade feel.

Player Health System

The **PlayerHealthController**
handles health UI, hit feedback, and (in non-“SecondLife” contexts) death behaviour.

Core Responsibilities

HP Bar Sprite Mapping:

- Health percentage is discretised into multiple sprite states for a retro-style segmented health bar.

Hit Flash + SFX:

- Incoming damage triggers a temporary red overlay with randomised impact sounds.

Events via DamageReciever:

- All health changes originate through **DamageReciever** (used by both player and enemies), keeping damage logic consistent across the game.

Design Rationale

Although the player cannot truly “die” due to the Second Life system, the HealthController still communicates tension and damage in real time, which preserves the illusion of danger.

The Second Life System (Accessibility & Low Friction Design)

The **PlayerSecondLifeController**
is a major QoL and accessibility feature designed after early user feedback. It replaces traditional death with a *cinematic fail-state* that fully restores the player while maintaining immersion.

Why It Exists

- Testers expressed difficulty managing enemy density and bullet visibility in early builds.
- We wanted to avoid excessive balancing complexity and keep the experience *approachable* for less skilled players.
- The system preserves the *illusion* of danger while almost never punishing the player.

Sequence Overview

When health reaches zero:

Game Paused Invisibly:

- Uses a custom pause method so UI is hidden but all systems that rely on the global pause flag behave correctly.

Hitstop and Camera Shake:

- A short hitstop freezes action, followed by dramatic shake to emphasise impact.

Fade to Black + Death SFX:
A timed fade provides a smooth transition into the “downed” state.
Black-Screen Audio Cue:
Prevents abrupt silence and enhances atmosphere.
Full Restore:
Player HP is reset to maximum, and ammo supplies for each weapon type are replenished.
Resurrection Fade-in + Flash:
A long fade-in synced with resurrection audio restores visibility, followed by a brief white flash for energy.
Control Restored:
Camera and movement scripts are re-enabled, time resumes, and gameplay continues as if nothing happened.
Design Rationale
The Second Life system keeps the game fast-paced and comedic while preventing frustration. It also fits thematically into the absurd tone of the overall experience.

Footstep & Jump Audio System
The **PlayerFootstepAudio** script
links player motion states with adaptive sound playback.
Key Features
Grounded vs. Airborne Detection:
- Landing → footstep timing resets, taking cues from **FirstPersonController.IsGroundedNow**.

Dynamic Timing:
- Sprinting reduces step interval values to create a faster cadence.

Pitch Randomisation:
- Variation ensures footstep audio never becomes repetitive.

Jump SFX:
- Triggered when transitioning from grounded to ungrounded.

Design Rationale
Audio is a major contributor to game feel. The system adds kinetic energy to movement, reinforces sprint state, and feeds directly into the “juice-first” design philosophy.

Integration With Game Feel & Juicing
Although player juicing is covered separately, it’s important to note that movement-state signals feed several other systems:
- **SprintParticleSystem** adds speed trails when sprinting.
- **CameraKickAndShake** responds to firing and impacts.
- **HitStop** pauses time briefly on weapon hits and deaths.
- **TrickshotSpinTracker** uses the camera orientation to detect 360° spins.
- **AuraFloaterUI** and score systems position text based on camera projection.

The movement, camera, and second life systems therefore act as *inputs* into the juicing framework, ensuring nearly every action the player takes produces responsive feedback.

Summary
Together, the player movement, health, and second-life systems provide:
- **Strong core feel** (tight movement, expressive camera).
- **High accessibility** (impossible to hard-fail, but feels like you can).
- **Continuous feedback** (footsteps, damage flashes, motion-based juicing).
- **Smooth integrated behaviour** with weapons, AI, scoring, and UI layers.

This suite of systems ensures the player experience is fast, forgiving, and visually/audibly rich, perfectly aligned with the project’s design.

The game features a unified **Aura Points System** that rewards the player for kills, trickshots, NPC dialogue rewards, and world collectibles. Aura acts as the game's "style currency," reinforcing the core pillars of spectacle, exploration, and exaggerated feedback.
The system consists of three major layers:

1. **Aura accumulation & multipliers**
2. **Visualisation of earned points** (floaters, kill-chain UI, collectible popups)
3. **World collectibles** with four tiers of value

These systems are connected through events, making the entire feedback and scoring pipeline modular and reactive.

Aura Points Manager (Scoring Logic)
(AuraPointsManager.cs)
The **AuraPointsManager** handles total point accumulation, UI updates, and the dispatch of aura events.

- Every kill triggers a call from **KillChainManager** into the scoring system, passing the world position of the death.
- The system applies **tiered multipliers**, determined by the kill chain state. Each tier multiplies the base kill value cumulatively, which increases sharply during high streaks.
- (Handed by tierMultipliers inside AuraPointsManager)
- After applying the multiplier, the manager updates the on-screen aura counter and sends a **points-added event**.

The event system ensures that UI floaters, trickshot systems, and score-driven feedback operate independently of gameplay logic.
In addition, the manager exposes a **bonus point function** for arbitrary rewards such as dialogue bonuses or collectibles.
This function integrates with the floater UI by automatically spawning text at the world position provided.

Aura Floater UI (Visual Feedback Layer)
(AuraFloaterUI.cs)
The **AuraFloaterUI** is responsible for turning numerical point gains into **floating visual indicators** that appear near the location where the points were earned.
Its behaviour includes:

- Converting world positions to canvas coordinates
- Spawning a temporary text element
- Applying a vertical drift motion
- Fading the text out and self-destroying

This system is triggered automatically through the AuraPointsManager event pipeline:

- Standard kills → floaters appear with "+X"
- Trickshot kills → floaters appear with custom text such as "TRICKSHOT +50"
- Collectibles → floaters appear conditionally depending on label usage

The floater system is deliberately **over-the-top** and visually loud, reinforcing the arcade nature of the scoring feedback.
(AuraFloaterUI is referenced throughout by AuraPointsManager via its static calls.)

Kill Chain Integration (Combo-Based Multiplier System)
(KillChainManager.cs)
The **KillChainManager** manages streaks, combo windows, and kill tiers.

- Every enemy death increments a streak counter and refreshes a short combo timer.
- When streak thresholds are crossed, the kill tier escalates (e.g., *RAMPAGE → MAYHEM → GODLIKE*). This is accompanied by UI animation, screen shake, and audio.

- (Tier thresholds and names are defined within KillChainManager.)
- The kill chain state directly feeds into AuraPointsManager, increasing the multiplier applied to kills.
- This system aligns scoring with spectacle-players are encouraged to chain kills and maintain momentum for maximum aura gain.

Collectibles System

Collectibles serve as the **exploration-based scoring path** and also reinforce the absurdity/comedy theme with dramatic pickup popups and audio.

Collectible Items (World Objects)

(CollectiblePickup.cs)

A collectible item:

- Bobs up and down, rotates, and optionally billboards to the player camera, making it visible from a distance.
- Contains identity data (ID, sprite, display name, aura value).
- On trigger interaction:
  - Registers itself with the persistent **CollectibleTracker**
  - Displays a large-screen popup via **CollectiblePickupUI**
  - Plays pickup SFX and VFX
  - Awards **silent aura** (no floater unless a label is provided) using AuraPointsManager

Collectible Tracked Persistence

(CollectibleTracker.cs)

To prevent repeated farming, collectibles register themselves through a global static tracker.
Once collected:

- The ID is added to a HashSet
- Future pickups of the same ID are ignored

This supports session persistence and ensures that collectibles function as intentional, one-time rewards.

Collectible Pickup UI

(CollectiblePickupUI.cs)

When a collectible is acquired, a **large, stylised popup** appears at the bottom of the screen:

- Displays the icon
- Displays the item name
- Displays the aura awarded
- Animates upward over time and fades out (handled through a coroutine)

This popup system is independent from the aura floater system and is purposely **more theatrical**, emphasising exploration rewards compared to combat rewards.

Summary

The game's points system is built around **high feedback density, modularity, and player empowerment**. The flow can be summarised as:

1. **Enemy killed → KillChainManager updates streak & tier → AuraPointsManager applies multiplier → AuraFloaterUI displays feedback.**
2. **Collectible found → CollectiblePickup triggers UI popup + aura reward → Tracker prevents duplicates.**
3. **Trickshot or special reward → AuraPointsManager.AddBonusPoints → Custom floater text.**

This unified pipeline ensures that every kill, discovery, and stylish action generates meaningful, visible feedback.
The four collectible tiers add pacing, exploration value, and allow designers to reward curiosity without impacting combat balance.

The project uses a layered UI architecture consisting of:

1. **Title Screen & Main Menu**
2. **In-game HUD & Pause Menu**
3. **Collectible/KillChain notifications and aura floaters**
4. **Loading and Credits screens**

All UI systems are deliberately animated, audio-reactive, and visually "juiced" to reinforce the game's expressive, retro-arcade identity. Unity's UI system is extended with custom scripts, coroutines, audio blending, and animation utilities.

Title Screen & Main Menu UI

Title Screen

(TitleScreenController.cs)

The title screen is intentionally simple: any key press transitions into the main menu using a **flash-to-white effect**, synchronized SFX, and a temporary shutdown of idle animations. The controller:

- Plays a **flash overlay animation** with a short pulse of brightness.
- Stops title screen music instantly for dramatic impact.
- Supports **two separate SFX layers** (flash sound + optional extra key-press SFX).
- Fades out after SFX finishes, preventing hard cuts.
- Loads the Main Menu once the visual/audio sequence finishes.

This creates a snappy, stylised transition that sets the tone for the game's exaggerated presentation.

Main Menu

(MainMenuController.cs)

The main menu includes several juicing systems:

- **UI pulse/float animations** (using *UIPulseAndFloat*, see below) for idle attention-grabbing.
- **Flash overlay** on every button press.
- **Music fade-out** during transitions to avoid abrupt silence.
- **Global and per-button SFX:**
  - Each button triggers a shared confirmation sound.
  - Additional bespoke SFX can be attached via *ButtonSFXTrigger.cs*
- **Scene transitions** that wait for UI flash + SFX before loading.

The intent is to make even menu navigation feel tactile, energetic, and aligned with the absurdist tone of the rest of the project.

Menu Parallax Backgrounds

(MenuParallax.cs)

Menu backgrounds use a subtle **parallax drifting system**:

- Smooth oscillating position offsets
- Different drift patterns for layered depth
- Non-intrusive movement that gives the UI a retro "living wallpaper" feel

This reinforces visual identity without distracting from navigation.

UI Pulsing and Floating

(UIPulseAndFloat.cs)

This is a reusable UI animation component applied across title/menu elements. It offers:

- **Alpha pulsing** (fading in/out)
- **Vertical floating** (sinusoidal motion)

- **Scale pulsing**
- **Rotational oscillation**
- **These animations run on unscaled time, so menu feedback is unaffected by pause states or slow-motion effects elsewhere in the game.**

Pause Menu

PauseMenuController

(PauseMenuController.cs)

This is the most complex UI subsystem. It synchronises game state, audio, and UI behaviour when pausing/unpausing.

Key Features

1. **Global Pause State:**
   - Sets Time.timeScale = 0f.
   - Broadcasts a static IsPaused flag used by movement, shooting, AI, and juicing systems.
2. **Audio Capture & Restoration:**
   - Stores the playback time and playing state of all active AudioSources.
   - Stops everything except:
     - Pause music
     - Radio system music
     - Temporary UI SFX (identified with a marker component)
   - Restores all audio exactly where it left off when unpausing.
3. **Pause Music & Radio System:**
   - Default pause music plays when the menu is opened.
   - Clicking anywhere on the menu selects a random “radio track” from a configured library.
   - Tracks avoid immediate repeats.
4. **Cursor Control:**
   - Unlocks cursor for menu usage, relocks it when leaving.
5. **UI Panels:**
   - Supports a restart confirmation panel.
   - Plays button click SFX before transitions.
6. **Integration With Second-Life System:**
   - Exposes ForcePauseWithoutUI() and ForceUnpauseWithoutUI() for the **PlayerSecondLifeController**, ensuring death/resurrection phases reuse pause logic without showing the visible menu.

This system ensures the game can be paused at any moment with no audio desync, and the radio mechanic adds a chaotic flair consistent with the game’s humour.

In-Game HUD & Gameplay UI

The HUD (weapon info, ammo, crosshair, aura meter, etc.) was documented earlier under Weapons & Player Systems, but here is the UI-specific framing:

- **Ammo & weapon UI** comes from *FirstPersonUIController*, updating each frame and reacting to weapon switches.
- **CrosshairController** colours the reticle based on whether a shot would deal valid damage.
- **KillChainManager UI** shakes and flashes based on kill tier.
- **AuraPointsManager** shows total aura with an embedded sprite icon.
- **AuraFloaterUI** spawns floating score popups in screen space for kills, trickshots, and collectibles.

This UI layer is entirely reactive, no logic lives here, all behaviour is driven by scoring, damage, or event systems.

Collectible & Pop-Up UI
Collectible-specific UI is triggered via **CollectiblePickupUI**
when a player picks up a world collectible. It:

- Creates a temporary popup with the collectible's icon/name/aura value
- Fades upward over time
- Uses a CanvasGroup fade for smooth transitions
- Deletes itself automatically via a coroutine

This system intentionally stands out from combat-related floaters to emphasise exploration rewards.

Credits Screen
CreditsSceneController
(CreditsSceneController.cs)
The credits screen adds an additional layer of polish:

- **Fade-in** of the entire screen and **music fade-in** when entering
- **Button-triggered back transition** using the same button click + optional SFX system
- **Randomised click-anywhere SFX** for chaotic expressiveness
- **Fade-out** of music and UI before returning to the main menu

This gives the credits an actual sense of presence rather than a static end screen.

Loading Screen
LoadingScreenController
(LoadingScreenController.cs)
The loading screen is intentionally minimal: a short mandatory pause followed by a scene transition.
This is primarily for pacing and allows the menu transition SFX and flash overlays to finish playing cleanly.

UI Audio Layer (Global SFX Behaviour)
Across the entire UI stack:

- Buttons trigger a **global confirm SFX** or optional per-button extras (via *ButtonSFXTrigger.cs*).
- Menu transitions are synchronised with audio fade-outs, preventing harsh cuts.
- Pause menu radio tracks add humorous unpredictability.
- Click-anywhere SFX on the credits screen encourage playful interaction.

This consistent audio design keeps UI interactions lively and coherent with the game's overall juiced aesthetic.

Summary
The project's UI systems are built around:

- **Strong audio presence** (button SFX, music fades, randomised clicks, radio tracks).
- **High visual juicing** (flashes, pulses, floats, parallax, transitions).
- **Consistency across scenes** (unified flash overlays, fade timings, motion patterns).
- **Event-driven architecture** for responsiveness and modularity.
- **Playful, expressive feedback** matching the game's chaotic tone.

From the title screen all the way to the pause menu and credits, the UI contributes not only to usability but also to the game's identity, humour, and sense of momentum

The game culminates in a dedicated boss encounter that functions as both a mechanical climax and a narrative punchline to the game's absurd tone. The boss system is composed of three interconnected components:

1. **BossActivationController** – controls when and how the boss appears
2. **BossController** – handles all AI behaviour, stage transitions, healing, and attacks
3. **BossSpriteController** – drives all boss visuals, animation states, and pulse effects

Together, they create a multi-phase fight with cutscene-like transitions, dynamic difficulty escalation, and a fully juiced presentation.

1. Boss Activation & Intro/Outro Flow

BossActivationController

The boss is not active from the start. Instead, the game waits for the player to earn **1,000 aura points**, which ties the boss trigger directly into the scoring, exploration, and kill-chain systems. This ensures that players engage with the full game loop before the final encounter.

Once the threshold is reached:

- **The entire boss hierarchy is enabled** (previously disabled to reduce overhead).
- The player receives a **large ammo injection** for their pistol to ensure they are fully stocked for the fight.
- A custom **intro panel** fades in with SFX, functioning as a lightweight cutscene.
- A listener is added to the boss' **DamageReciever.onDeath** event, linking the boss' death directly to the outro and ending sequence.

After the boss dies:

- An **outro panel** fades in with its own SFX, holding briefly.
- Controls are disabled and cursor unlocked.
- The game loads the **CreditsScene**, formally ending the vertical slice.

This controller is the "glue" between gameplay systems (scoring, ammo, player readiness) and UI/transition systems (panels, fades, scene loads).

2. Boss AI, Behaviour, and Stage Logic

BossController

The boss is the most complex AI in the project, following a **two-stage design** separated by a **healing intermission**.

Common Behaviour Across Both Stages

- Always faces the player horizontally.
- Movement uses Rigidbody-based translation for consistency with other enemies.
- Pauses AI whenever:
  - The player is in second-life respawn
  - The game is paused
  - The boss is stunned
- Uses the same **DamageReciever** system as all other enemies, giving full compatibility with weapon types, immunities, and hit reactions.

Stage 1: Ranged Phase (Gunfire)

In Stage 1, the boss behaves like an elite sniper-style enemy:

- Moves at moderate speed
- Telegraphed warmup before firing
- Fires custom projectiles at the player
- Uses Stage 1-specific attack SFX with random pitch variation
- Sprites indicate walking, engaging, attacking, and taking damage

This makes the fight initially readable and survivable for newer players.

Mid-Fight Heal Sequence
When the boss reaches **50% health for the first time**, a **healing cutscene** plays:

- Boss enters a special sequence, halting AI
- Plays the “drink” animation via *BossSpriteController*
- Plays drink SFX with pitch variation
- Fully restores Stage 1 health
- Reapplies the low-health tint system

This creates a comedic pause in the fight and reinforces the cartoonish aesthetic.

Stage Transition (Transform Sequence)
When health again reaches **50%**, the boss transforms into Stage 2:

- Starts “transform” animation
- Plays transformation SFX
- Scales up the boss’ size dramatically
- Switches sprite sets to Stage 2 sheet
- Clears and reassigns damage immunities:
  - Now immune to **rifle** and **explosive** damage
  - Encourages weapon switching and variety
- Fully resets health to the new Stage 2 maximum
- Exits into the melee-focused phase

This escalation is both mechanical (forcing close-range fighting) and aesthetic (giant boxer transformation).

Stage 2: Melee Boxing Phase
In Stage 2, the boss switches from shooting to an aggressive melee moveset:

- Moves significantly faster
- Uses high-damage punches with low cooldown
- Alternates left/right punch sprites
- Continues using pulse-based SFX
- Uses Stage 2-specific animation set from *BossSpriteController*

This creates a frantic endgame where spacing and movement matter more than aim.

Boss Visual Controller (Sprite, Animation, Pulsing)
BossSpriteController
The sprite controller handles all animation states for both stages:

- Walking loops (2-frame retro flipbook)
- Engaging/telegraph frames
- Attack pulses
- Hit pulses
- Special animations (drink, transform)
- Stage-specific low-health variants

The controller uses coroutine-driven sprite swaps and a pulse timer that prioritises high-impact frames (attack, hit) over idle animations. This makes the boss feel reactive and expressive without requiring traditional animator controllers.

4. Integration With Game Systems

The boss ties into nearly every major system documented earlier:

1. Scoring & Progression

The boss only appears once the player has engaged with the core loop enough to earn 1,000 aura points.

This ensures the boss acts as a *reward* for skillful play and exploration.

2. Weapons System

DamageType immunities change between stages, validating the multi-weapon system design and forcing meaningful weapon switching.

3. Player Second-Life System

The boss respects the global "PlayerIsRespawning" flag, its AI suspends itself until the player returns, preventing unfair hits during revival sequences.

4. Enemy Juicing Layer

The boss inherits hit flashes, tinting, knockback behaviour, SFX variation, and death VFX from shared components like:

- EnemyHealthTint
- EnemyFlash
- KnockbackOnHit
- EnemyAudioController

5. UI & Transition Systems

Boss intro/outro panels use the **same fade logic and audio blending** principles as:

- Title flash transitions
- Menu flash overlays
- Credits fade sequences
- Loading screen pacing

This keeps presentation consistent across the entire game.

6. Ending the Game

The boss' death event triggers the **CreditsScene** via the fade-out panel, formally completing the play session and providing a clear end to the vertical slice.

Summary

The boss system acts as the final expression of every major mechanic in the project:

- **AI complexity** (warmups, projectiles, melee logic, stun blocking)
- **Juicing systems** (sprites, pulses, audio, scaling, transitions)
- **Scoring & progression** (aura threshold to activate)
- **Weapon variety** (damage immunities force switching)
- **Player survivability** (integrated with second life)
- **Cinematic UI transitions** (intro/outro sequences)

It is intentionally exaggerated and comedic, aligning perfectly with the game's absurd, high-energy tone while providing a clear and satisfying endpoint for the vertical slice.

Use of AI in the Development Process

Artificial intelligence tools played a supportive but important role throughout the development of both the game and this report. AI was never used to generate full systems or write complete scripts in isolation; instead, it served as an **auxiliary problem-solving and refinement tool**, comparable to a highly responsive technical reference or a real-time "rubber ducking" assistant. All gameplay systems, mechanics, and architectural decisions were implemented manually, with AI participation limited to clarification, debugging guidance, refactoring support, and editorial assistance.

AI Support in Coding and Technical Development

Throughout the project, AI was used primarily as a **development aid**, assisting in the following areas:

- **Rubber ducking:** explaining my intended logic or problem to the AI often helped me identify mistakes or oversights in my own reasoning.
- **Refactoring suggestions:** once systems were already implemented, AI was used to recommend cleaner structure, reduced duplication, or safer control flow measures.
- **Safeguard proposals:** the AI frequently highlighted areas where additional checks (e.g., null guards, pause-state checks, state validation) could prevent runtime issues.
- **Bug reproduction & interpretation:** by describing symptoms and providing relevant snippets, the AI helped interpret error logs and identify likely causes.
- **Mechanic improvement suggestions:** suggestions for enhancements to juicing, NPC UX, camera responsiveness, or modularity helped refine the final experience.
- **Resource compilation:** the AI was used to summarise or point towards relevant Unity documentation or known best practices when exploring new techniques.

At no stage were complete gameplay scripts authored by AI. Instead, AI assistance was used **only after writing the initial logic**, typically in a rough or haphazard form.

Example – Coding Usage

When implementing the Second Life system, Jibril would first write a functional draft that restored the player on death. AI was then used to:

- propose a safer sequence order (freeze → fade → revive → unfreeze),
- suggest additional pause-state safeguards,
- and recommend comment structures to clarify the timing of callbacks.

The final implementation remained his own code, but benefited from conceptual clarification and reinforced defensive programming.

AI Support in Report Writing

AI was also used to refine the written report in a similar manner:

- **Drafting conceptual clarity:** after writing a section describing a mechanic, we used AI to identify unclear phrasing or missing context.
- **Structural clean-up:** AI helped reorganise long explanations into coherent paragraphs or subsections without altering their substantive meaning.
- **Tone and consistency:** the report went through iterations where content was rewritten, and AI assisted in smoothing inconsistencies in tense, voice, and academic tone.
- **Summarisation:** when dealing with large blocks of text, AI helped condense ideas without losing technical depth.

Example – Report Usage
Jibril writes a complete description of the enemy activation system, including activation radii, chase behaviour, and warmup mechanics. AI was used afterwards to:

- tighten the wording,
- ensure the terminology matched earlier sections,
- and suggest a more logical ordering (activation → movement → attack → integration).

Again, the underlying content, concepts, and technical descriptions originated from my own development notes and understanding; AI functioned as an editorial assistant rather than an author.

Summary of AI Contribution
Exclusively using **ChatGPT**, AI supported the project in the following ways:

- Rubber ducking for debugging, logic checking, and architecture reasoning.
- Suggestions for gameplay refinements and juicing enhancements.
- Refactoring advice and clarifying comments for complex scripts.
- Identifying potential edge cases and recommending extra safety guards.
- Assisting in interpreting error logs and diagnosing likely faults.
- Collating or summarising technical resources and Unity documentation.
- Editing, tightening, and restructuring sections of written report content while leaving the technical substance unchanged.

AI was *not* used to generate full systems, asset pipelines, or game logic from scratch. All gameplay code, design decisions, and mechanical implementations were manually authored.

TESTING AND ANALYSIS

INTRODUCTION

This section outlines how our game was tried and tested, going through changes after changes to reach this final vertical slice. The chapter contains details about major testing sessions conducted both offline and online. The final UI testing was conducted among peers at the Nexus Games Conference 2025, held at Google, Ireland, and the final Playtest - Combat Simulation was conducted among students of Media at the TU Dublin Grangegorman Campus.

Session 1 - July

Venue: Yugo Student Accommodations
Prototype: Figma
Purpose: To test the UI for feasibility
Participants: 3
Task: Load an already saved game

Questions:

Was the UI easy to understand at first glance?

- Yes / Somewhat / No

Was the text (HUD + dialogue) clear and easy to read?

- Yes / Somewhat / No

Did any UI element feel confusing or distracting?

- No / A little / Yes (please specify)

Were the UI controls and prompts easy to follow?

- Yes / Somewhat / No

How would you rate the UI overall?

- Great / Good / Okay / Needs improvement

Findings

- The UI seems functional, but not that clear
- Everyone got the HUD layout, but 2 of them struggled with the load and save feature
- Interaction prompts disappeared quickly, so they were missed by all the first time
- Dialogue was readable
- Stats and counts were missed because of size issues

Inference

The UI is okay for gameplay, but it needs to be clearer for it to be attractive to users. The buffer time and colour contrast of interaction cues and dialogue boxes should be increased so that the users get time to read. Important HUD elements and stat bars should not disappear with motion. A small UI tutorial should be added to make it easy to interact.

Session 2 - October

Venue: Nexus Game Conference, Google Ireland
Prototype: First build, Figma UI
Participants: Industry experts (Game Developers, Game UX Designers)
This was indeed a special type of testing, as our testers were industry experts, so instead of our asking questions to them, they gave us feedback on what they thought of the game as a whole.

- A game UX designer asked me to get rid of the options and controls from the pause menu and the main menu for the demo, as it creates clutter, and since it doesn't serve any purpose in the game right now, it is absolutely not necessary.

- Another indie game artist suggested changing the art style from vector to retro arcade to match the vibe.

- A game developer strictly warned us not to plan more than one mission

Out of all the people who saw our game, one thing was for sure: they all loved the idea, and it needed some tweaking. Just as they said, the main menu just shows "PLAY, CREDITS & EXIT", and the pause menu has "RESUME, RESTART & EXIT." We cut down the 3 missions and side quests to 1 mission and collectibles, to reduce the workload for this vertical slice. The art style was changed much later.

Session 3 - November

Venue: TU Dublin Grangegorman Campus
Prototype: Combat Simulation Playtest
Participants: School of Media Students

Participant Overview
Total Participants: 11

Experience With Genre
Very familiar: 3
Somewhat familiar: 5
Expert: 1
Not familiar: 2

Playtime Duration
Under 5 minutes :Majority
5–10 minutes :Several
10+ minutes :Few

Most participants only played briefly, suggesting the build supports short sessions or that players reached core content quickly.

Controls & Ease of Understanding
How intuitive were the controls?
The majority selected "Easy to understand."
A few reported initial confusion, especially related to:
- Navigation
- Understanding what to do next

Improvement Needs Mentioned
- Better onboarding/tutorial
- More clarity in controls and goals

Overall Enjoyment & Satisfaction

How much fun was the game?
Participants commonly reported:

- “Fun”
- “Very fun”
- “Slightly fun”

No one had a negative opinion on that.

Likelihood of playing again

- Mostly Likely or Very likely
- Only one participant marked Unlikely

This indicates the gameplay loop is engaging even in its early form.

Visuals, Sound, and Performance

Visual Appeal

Comments indicate:

- “Good” and “very good” visual design
- Suggestions for more character variations or building designs

Performance

Participants generally reported:

- Smooth performance
- No significant lag or crashes mentioned

Suggestions for Improvement

- More content, Additional buildings, more characters, expanded world
- Better animations. Requests for “smoother animations.”
- Improved onboarding. Several players didn’t understand the controls immediately
- Gameplay depth, Ability to run from enemies, more actions, clearer objectives
- Enemy behavior Refinements to interactions/AI suggested

Notable Individual Comments

- “Maybe some buildings or more character designs...good job!”
- “Maybe smoother animations.”
- “Ability to run away from the enemies.”
- “The onboarding should be changed; I didn't fully understand where to go.”
- “Fire game 🙏”

CONCLUSION

The testing process we followed was a bit rough, as this was a PC game and testing it required a proper setup, yet we managed to get it tested by multiple demographics. The changes made from the feedback of the final test have been mentioned in the ‘Testing & feedback-driven iterations’ section. Overall, the testing phase gave us an idea from almost all the perspectives of the industry as well as our target audience.

EVALUATION

INTRODUCTION

This evaluation examines the technical, artistic, and design quality of the final vertical slice, reflecting on how well the project met its goals, where compromises were made, and how effectively systems were implemented under the constraints of time, scope, and team size. It also critiques the production process, summarises discoveries made during real-world testing, and highlights future improvements.

Critical Success Factors

The project's success depended on delivering a vertical slice that:

- Demonstrated a responsive, juiced-up first-person combat system.
- Included enemy AI, weapons, dialogue, collectibles, UI, and a boss encounter, all working cohesively.
- Achieved a distinctive PS1/PS2-inspired visual style.
- Delivered gameplay that matched the "structured chaos" brief.
- Catered to both skilled FPS players and casual players through accessibility features (e.g., the Second Life system).

All these factors were successfully achieved in the final build.

Minimum Viable Product (MVP)

The MVP aimed to deliver a polished combat arena with:

- First-person movement, shooting, and trickshot systems.
- Functional enemy types (melee, mobile sniper, stationary sniper, boss).
- A small, explorable map with NPCs and collectibles.
- A reactive scoring/aura system.
- A working pause menu, UI, and transition flow.
- A complete boss fight tied to scoring progression.

Several initially planned features were cut to preserve the MVP timeline, including:

- A full options/settings menu.
- Advanced audio mixing using FMOD.
- Larger level expansions or mission systems.

The final MVP was stable and complete, providing a coherent and satisfying experience.

Code Critique

The codebase ultimately functions reliably, but it reflects the reality of a long production cycle with evolving priorities, knowledge, and coding style.

Strengths

- **Clear separation of data and logic** through ScriptableObjects (weapons, dialogue).
- **Loose coupling** through events, delegates, and shared entry points (DamageReciever, IHitReactable).
- **Robust global state handling** via pause flags, respawn flags, and shared managers.
- **Highly modular juicing pipeline**, enabling rapid iteration on feedback systems.

Weaknesses

- Many systems contain **single-layer safeguards** ("if paused, return;"), which provide stability short-term but are not scalable or elegant.
- Some subsystems became **tightly interwoven** (e.g., boss → player → pause logic), making refactoring more complex.
- Several scripts could be **condensed or merged**, especially UI or feedback scripts with overlapping responsibilities.

- Naming conventions and commenting style changed over the months, leaving inconsistent documentation in older files.
- Some core logic would benefit from state machines instead of boolean-driven behaviour.

In short, the codebase works reliably in its final form, but it is held together with “just enough” structure, adequate for a vertical slice, but vulnerable under expansion.

Code Log

Over the course of development:

- Early prototypes focused on movement, shooting, and AI pathing.
- Mid-development introduced significant overhauls: juicing systems, aura pipeline, collectibles, and NPC dialogue.
- Late development involved stabilising the boss fight, revising pause logic, and polishing UI transitions.
- The final weeks were dedicated to fixing obscure bugs, reducing friction for new players, and implementing fallback safeguards.

This iterative timeline produced a functional but organically grown codebase, where new features often required retrofitting older ones.

Design Critique

Art Direction

Early builds used low-poly 3D models, but the contrast between 3D assets and 2D elements was visually jarring. Switching to a blended PS1/PS2 retro aesthetic made the entire visual identity more cohesive. The reduced render scale flattened the 3D environment, allowing 2D sprites (enemies, weapons, UI) to sit naturally within the scene.

This stylistic pivot simplified asset production, increased consistency, and aligned perfectly with the absurd retro theme.

Audio

Because FMOD was removed from scope, audio implementation became ad-hoc:

- Many SFX are driven by individual scripts.
- Volume levels are inconsistent across devices.
- Several temporary audio sources ended up being permanent out of necessity.
- No global mix controls or platform-specific tuning were implemented.

Although chaotic, the final soundscape enhances the comedic, arcade tone, but it is not technically robust.

Technical Stability
The project includes numerous fallback measures to prevent the player from becoming stuck or frustrated:

- The **Second Life system** prevents hard failure while maintaining tension.
- The **boss**, **enemy AI**, and **movement systems** all respect global pause/respawn flags to prevent unintended behaviour.
- Generous ammo and health drops avoid balancing spikes.
- The level size is intentionally small so that restarting after a bug is low-friction.
- Randomisation (spawn points, audio) increases replayability but complicates systematic testing.

Despite the “held together with tape” nature, no major game-breaking bugs were observed in final testing.

Feedback to Users and Data Analysis
Playtest observations were instrumental in shaping the final version:

- Early players wanted more **guidance**, leading to the creation of the full NPC dialogue system.
- Aesthetic feedback triggered a **complete art overhaul**, which massively improved reception.
- Combat felt “weightless” early on; testers directly inspired the addition of **knockback**, **hitstop**, **impact SFX**, and **clear enemy telegraphs**.
- Difficulty variance among testers confirmed the need for the **Second Life system** and **exploration rewards**.
- Viewers during the Nexus Games Conference responded strongly to juiced UI and chaotic elements, reinforcing the game’s artistic direction.

Testing therefore guided not only improvements, but also confirmed that the “structured chaos” design philosophy was effective.

Teamwork and Production
Production responsibilities naturally divided according to strengths:
Jibril

- All gameplay programming (weapons, AI, juicing, scoring, boss).
- All UI implementation and menu systems.
- All audio sourcing, integration, and functional logic.
- Scene setup, iteration, bug-fixing, and build management.
- Final polish across all systems.

Jacob

- All 2D asset production (sprites, characters, UI art).
- Level layout planning and basic block-out.
- Initial environment modelling (later replaced or restyled).
- Support with visual direction and thematic consistency.

Collaboration was positive. Both members contributed to design decisions, testing sessions, and the artistic direction, resulting in a coherent final product.

Future Work

Given more time, the following improvements would meaningfully enhance the project:

Technical Improvements

- Replace boolean-driven logic with **formal state machines** (player, AI, UI).
- Rebuild the audio architecture using **audio mixers** or FMOD.
- Introduce object pooling, reducing instantiation overhead for effects/UI.
- Refactor juicing systems into a unified feedback architecture.
- Improve modularity by consolidating repeated logic across scripts.

Content & Design Expansion

- Add additional missions, maps, and enemy types.
- Implement a proper options menu (audio sliders, sensitivity, accessibility).
- Expand the dialogue system with conditional logic and NPC memory.
- Add upgrading, vendors, or skill-based progression.
- Expand boss behaviour with more phases, attacks, and environmental hazards.

Quality & Accessibility

- Normalise SFX levels and implement global mixing controls.
- Add UI scaling options.
- Improve onboarding through optional tutorials or guided missions.

Polish & UX

- Improve animation smoothness for sprites with interpolation or tweening.
- Add VFX for water distortion, bubbles, and underwater ambience.

Overall, the project provides a solid foundation for expansion, but significant refactoring would be necessary before scaling into a full game.

Conclusion

The vertical slice successfully demonstrates the game's identity: chaotic, expressive, humorous, and mechanically dense despite its deliberately absurd presentation. While the codebase is functional but messy, the systems work cohesively to deliver an entertaining and technically robust experience. The production journey, from early 3D experiments to the final retro-aesthetic FPS, shows clear design reasoning and adaptation based on user feedback. Future work could refine, expand, and stabilise the systems, but as a vertical slice, the project clearly meets the expectations of an MSc major project.

Art & Asset Sources

Stylised Nature Environment (CG Assets Pack).
Used for environmental models and foliage.
Sourced from Unity Asset Store / CG Asset collections (original pack title: *Stylized Nature Environment*).

Collectibles.
Collectible items were created through a combination of hand-drawn artwork and concepts provided by peers.

Audio Sources
OpenGameArt.org – Library of Game Sounds.
General-purpose SFX used for UI feedback, interaction sounds, and miscellaneous events.
Available at: https://opengameart.org/content/library-of-game-sounds

The Spriters Resource – Sound Effects Archive.
Used for stylised retro SFX to reinforce the game's PS1/PS2-inspired aesthetic.
Available at: https://sounds.spriters-resource.com/

Minecraft Wiki – Dolphin Sounds.
Referenced for NPC, enemy, and player vocalisations where animal-like or comedic SFX were required.
Available at: https://minecraft.wiki/w/Category:Dolphin_sounds

Minecraft Wiki – Fish Sounds.
Used to support fish-themed NPC and enemy audio design.
Available at: https://minecraft.wiki/w/Category:Fish_sounds

Khinsider Game Soundtrack Archive – Ridge Racer Type 4 OST.
Used as stylistic reference and as background music to match the late-90s arcade aesthetic.
Available at: https://downloads.khinsider.com/game-soundtracks/album/r4-ridge-racer-type-4

YouTube – Miscellaneous Ambience, Music, and SFX.
Used for additional atmospheric layers, environmental ambience, and non-licensed supplementary sound effects where permitted.

Primary playlist referenced:
https://www.youtube.com/watch?v=mqpgTLH9UP0

General Media & Inspiration Sources
YouTube.
Used for gathering ambience, stylistic references, and analysis of retro audio design techniques aligned with the project's aesthetic direction.

CONCLUSION

The development of iT AiN'T THAT DEEP demonstrates how a focused, iterative, and user-centred design process can successfully deliver a polished and expressive vertical slice within the constraints of an MSc major project. From the initial user need analysis through to the final implementation, each phase shaped a product that remains true to its core vision: a chaotic, humorous, and mechanically dense first-person experience driven by clarity, responsiveness, and player expression.

Across the project, significant design and technical pivots were made, most notably the shift from a 3D asset-heavy pipeline to a hybrid 2D/3D PS1/PS2-inspired style. This decision, paired with the "structured chaos" design philosophy identified early in the report, enabled rapid iteration and cohesive visual identity. The change also aligned naturally with the personalities and expectations of the target personas highlighted in the User Need Analysis (pages 9–10), giving the game a distinct aesthetic and tone
.
Technically, the project delivered a comprehensive suite of interdependent systems, including weapons, enemy AI, player survivability, trickshots, juicing, scoring, dialogue, UI, and the multi-phase boss encounter. These systems, although occasionally held together by pragmatic safeguards, form a coherent whole and demonstrate the capability to design and implement modular gameplay frameworks. The result is a functional, reactive, and engaging combat loop supported by a rich feedback ecosystem.

Testing at multiple points in development (including the Nexus Games Conference and TU Dublin sessions) directly guided improvements to clarity, difficulty balancing, visual cohesion, and audiovisual impact. These iterative refinements helped validate initial assumptions, particularly the importance of readable feedback, low-friction play, and exaggerated game feel, while also motivating the introduction of new systems such as contextual NPC dialogue, collectible incentives, the Second Life mechanic, and expanded UI/audio feedback.

The final vertical slice is therefore a product of both creative ambition and practical constraint. It delivers a clear sense of identity, expressive game feel, and a complete start-to-finish experience that encapsulates the project's goals. While the codebase and broader architecture would require refactoring before scaling into a full game, the existing systems form a strong foundation for future development.

In summary, iT AiN'T THAT DEEP succeeds as a compact but characterful demonstration of gameplay engineering, UX iteration, visual design, and collaborative production. It meets the requirements of the MSc Major Project, provides evidence of technical and creative competency, and offers a clear roadmap for continued expansion beyond this vertical slice.