H446 OCR Programming Project

*Eshan Fadi*

*Upton Court Grammar School*

*Centre Number: 51427*

*Candidate Number: 8052*

Abrupt Animus

Table of Contents

[Project Ideas 5](#_Toc182414549)

[Engine and Language Evaluations 6](#_Toc182414550)

[Unity – C# 7](#_Toc182414551)

[Unreal Engine – C++ 7](#_Toc182414552)

[Godot Engine – GDScript 7](#_Toc182414553)

[Python (using Pygame) 8](#_Toc182414554)

[RPG in a Box – Bauxite 8](#_Toc182414555)

[Chosen Idea and Justification 8](#_Toc182414556)

[Analysis 9](#_Toc182414557)

[Problem Identification 9](#_Toc182414558)

[Computational Methods 9](#_Toc182414559)

[Stakeholders 13](#_Toc182414560)

[Silly Australian ® - Henry Masters (Founder and CEO) 13](#_Toc182414561)

[DrunkDriving ™ - Amarveer Flora (CEO) 13](#_Toc182414562)

[Research 14](#_Toc182414563)

[Titanfall 2 14](#_Toc182414564)

[DOOM (2016) and DOOM Eternal 14](#_Toc182414565)

[Half Life (and other Source Engine games) 15](#_Toc182414566)

[Ultrakill 16](#_Toc182414567)

[Omori 16](#_Toc182414568)

[Essential Features 17](#_Toc182414569)

[Movement 17](#_Toc182414570)

[Walking/Strafing 17](#_Toc182414571)

[Sprinting 17](#_Toc182414572)

[Jumping 17](#_Toc182414573)

[Crouching (both toggle and hold) 17](#_Toc182414574)

[Sliding 17](#_Toc182414575)

[Wallrunning 18](#_Toc182414576)

[Grappling 18](#_Toc182414577)

[Double Jump 18](#_Toc182414578)

[Boosting 18](#_Toc182414579)

[Combat 18](#_Toc182414580)

[Fire 18](#_Toc182414581)

[Zoom 18](#_Toc182414582)

[Swap weapon 19](#_Toc182414583)

[Quick melee 19](#_Toc182414584)

[Pick up weapon 19](#_Toc182414585)

[Reload 19](#_Toc182414586)

[Sound Effects 20](#_Toc182414587)

[Saving 20](#_Toc182414588)

[Multiple Levels 20](#_Toc182414589)

[Glitch 20](#_Toc182414590)

[Scoring System 21](#_Toc182414591)

[Merit 21](#_Toc182414592)

[Dialogue System 22](#_Toc182414593)

[Death Screens 22](#_Toc182414594)

[Graphical User Interface (GUI) 22](#_Toc182414595)

[Limitations 23](#_Toc182414596)

[Solution Requirements 23](#_Toc182414597)

[Hardware and Software Requirements 25](#_Toc182414598)

[Success Criteria 26](#_Toc182414599)

[Design 27](#_Toc182414600)

[Decomposition 27](#_Toc182414601)

[Combat 27](#_Toc182414602)

[Weaponry 27](#_Toc182414603)

[Swapping 27](#_Toc182414604)

[Guns 27](#_Toc182414605)

[Melee 27](#_Toc182414606)

[Grenades 28](#_Toc182414607)

[Player 28](#_Toc182414608)

[Movement 28](#_Toc182414609)

[Horizontal 28](#_Toc182414610)

[Vertical 28](#_Toc182414611)

[Wallrunning 28](#_Toc182414612)

[Entities 28](#_Toc182414613)

[Drones 29](#_Toc182414614)

[Soldiers 29](#_Toc182414615)

[Tanks 29](#_Toc182414616)

[Civilians 29](#_Toc182414617)

[Environment 29](#_Toc182414618)

[Walls 29](#_Toc182414619)

[Slopes 29](#_Toc182414620)

[Pickups 29](#_Toc182414621)

[Interacts 29](#_Toc182414622)

[UI 29](#_Toc182414623)

[HUD 29](#_Toc182414624)

[Menu 29](#_Toc182414625)

[Main Menu 30](#_Toc182414626)

[Options 30](#_Toc182414627)

[Saves 30](#_Toc182414628)

[Load game 30](#_Toc182414629)

[Dialogue 30](#_Toc182414630)

[Interpreter 30](#_Toc182414631)

[Actors 30](#_Toc182414632)

[Managers 30](#_Toc182414633)

[Structure Definition 31](#_Toc182414634)

[Algorithms 31](#_Toc182414635)

[Usability 31](#_Toc182414636)

[Variables 31](#_Toc182414637)

[Data Structures 31](#_Toc182414638)

[Classes 31](#_Toc182414639)

[Validation 31](#_Toc182414640)

[Iterative Tests 31](#_Toc182414641)

[Post Tests 31](#_Toc182414642)

[Development 32](#_Toc182414643)

[Evidence 32](#_Toc182414644)

[Input Detection 32](#_Toc182414645)

[Movement 32](#_Toc182414646)

[Look 32](#_Toc182414647)

[Collision Detection 33](#_Toc182414648)

[Entities (Damage) 33](#_Toc182414649)

[Interacting with weapons 34](#_Toc182414650)

[Dialogue 35](#_Toc182414651)

[Saving and Loading System 41](#_Toc182414652)

[Prototyping 45](#_Toc182414653)

[Modules 45](#_Toc182414654)

[Comments 45](#_Toc182414655)

[Naming 45](#_Toc182414656)

[Validation 45](#_Toc182414657)

[Reviews 45](#_Toc182414658)

[Development Testing 45](#_Toc182414659)

[Evidence 45](#_Toc182414660)

[Fails 46](#_Toc182414661)

[Evaluative Testing 46](#_Toc182414662)

[Evidence 46](#_Toc182414663)

[Usability Tests 46](#_Toc182414664)

[Evaluation 46](#_Toc182414665)

[Success Criteria 46](#_Toc182414666)

[Usability Features 46](#_Toc182414667)

[Limitations 46](#_Toc182414668)

[Maintenance 46](#_Toc182414669)

[Bibliography 46](#_Toc182414670)

# Project Ideas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Idea | Short Explanation | Complexity | Language | Computational Methods |
| Movement first-person-shooter game | A game where the player must simultaneously manage their momentum and there aim | Medium - ample use of math and GUI. Movement must be fine-tuned along with movement-based bug fixes. Player physics, and bullet physics, enemy AI, and modular interconnected components, OOP, libraries | C#, C++, GDScript, Python, easier with engines | OOP, database, modular design, abstraction for the GUI, reusability for functions |
| Game engine | An application designed to help and aid someone in the creation of game or another app | Difficult - lots of math, OOP, GUI, implementing a physics engine, along with functionality for any component, must include compilers and built in libraries to assist in building the game | C#, C++ | OOP, decomposition of functions |
| Programming Language | Writing a language and run it either being compiled, or to interpreted | Easy - filled with trees and if statements, and exception raising, must have built in functions, handle memory and add basic data structures | C#, C++, CPython | Selection, trees, decomposition, and modular design for various parts of the compiler |

## Engine and Language Evaluations

The purpose of an engine within software development is to speed up the development speed. Usually, these contain tools that manage and automate significant background processes, such as the script execution order, a built-in physics engine, and a lighting pipeline. Many modern engines feature a modular design for their workflow, with there being objects which you can attach components to, to add further functionality to the object. Some of these may be colliders, materials, or even scripts.

### Unity – C#

Released in 2005, Unity is an extremely useful engine that has great capabilities for both 3D and 2D, with powerful tools to assist in the creation of apps (mostly games) and experiences. It has a huge community surrounding it, making it incredibly easy to learn, with a wide variety of plugins to aid in the development of certain aspects of the process. The currently supported language used within it is C#, part of the C family. The language is quite easy to learn, having aspects like Python and JavaScript. It also simultaneously has the speed and access of C++, as it is a compiled language. I have some previous experience in using both Unity and C#, but the majority of their functionality is still to be explored. However, Unity recently made some economic decisions that have severely damaged the reputation of the company and software. As such, if I wish to continue the project afterwards, I may end up having to completely transfer the entire project to another piece of software, so this may not be the best long-term solution.

### Unreal Engine – C++

All the way from 1998, this engine is well known for boasting incredible visuals without much effort required from the developer. It is comparatively easier to optimise the application to run faster, as certain mesh optimizations exist natively to Unreal Engine 5 (namely Nanite). Proven its ability repeatedly, the Unity scandal has brought more eyes upon this amazing piece of software. However, due to its fabulous visual capabilities by default, making lightweight applications and games is far tougher, as many layers must be stripped away. Furthermore, it is well known that Unreal Engine is not built for the 2D experience, limiting the type of content that it can create. The language of choice for UE5 is C++, a language notorious for being extremely difficult to learn with low-level concepts such as pointers mixed in with high-level abstractions such as object-oriented programming. I have little-to-none prior experience programming with C++, so the learning curve to use UE5 is far greater than the other options.

### Godot Engine – GDScript

Godot is the game engine equivalent to Blender 3D in the 3D modelling world - a completely free and open-source alternative, which is only getting better and better every update, and will most likely become the industry standard within a decade. It uses a node-based system to construct the world and all objects which is similar to the systems that other game engines use. This intuitive system allows for more flexibility, greatly helping in the learning process. Godot mainly uses GDScript, a custom language which the engine supports completely for all platforms. It also supports C# and C++, but C# is not currently available to be compiled for the web in Godot 4.2 (the latest version of Godot at the time of writing). Godot can also support any other language using extensions to the engine. The worthwhile option would be GDScript, which is similar in syntax to JavaScript and Python, and has many built in functions such as is\_on\_floor() to handle common and simple checks automatically. The number of tutorials that the community and the Godot team themselves have made is vast and is only increasing, as many people flock from Unity to Godot, making it so that learning the engine and its language would be far easier than before.

### Python (using Pygame)

Python and Pygame together allow anyone to easily pick up and create anything and everything that they want. Unlike the other entries on this list, this is not an engine, which means that the creator must program in every feature, from collision handling and response, to manually testing individual locations for objects, instead of placing them in a visual scene and moving them easily. This significantly slows down the development of the application and can be extremely tedious as everything must be done by the developer, reducing the amount of creativity and passion that can be put into the project. While the language itself is extremely easy to learn, the underlying concepts behind game development such as matrix transformations and quaternions are extremely difficult to understand and waste one’s time when implementing them in such a small timescale. Such a project would be better suited to one with a far longer time frame to work within.

### RPG in a Box – Bauxite

Made using Godot, RPG in a Box is a game engine that has everything built within it for you: a voxel editor, fully working dialogue system, a sound effects generator, and even a visual scripting language. While many of the other engines on this list also have visual scripting as part of their arsenal, none use it or recommend it as much as RPG in a Box. It also has a secondary fully working scripting language called Bauxite, which is extremely similar to Lua, a famous programming language that has been used to make amazing games such as Hades. The engine is designed to be as friendly to a normal person as possible, such that no prior experience is required to even begin making something within this engine, but this also happens to be its downfall. Since so many features are required to use the built in system to have full functionality, it becomes harder to implement higher level concepts without effectively writing your own engine.

## Chosen Idea and Justification

The game engine is far too complex and time consuming for the constraint of time, while also requiring more learning than either of the other options. The programming language on the other hand, is simpler and would not allow me to easily incorporate as many computational methods. The movement first-person-shooter game is a great medium between the complexity of the two other options, while also allowing me to incorporate a great deal of creativity and interesting implementations of the course content. Furthermore, I will be using Unity to implement this since I have prior experience using the engine and C# as well, so I can spend more time focusing on constructing the project than learning how to use the program itself. I also get the benefit of using an engine which allows me to focus on unique ideas, rather than implementing standard algorithms and techniques.

# Analysis

## Problem Identification

An up-and-coming group of the gaming community is a collection of speedrunners, craving for fast movement and fluidity to beat the games faster and faster. To satiate this need, I have decided to construct a fast-paced movement first-person-shooter game, demanding both an understanding of momentum and precise aim, to hit a selection of enemies and reach the end of the level as fast as possible. The game will also have a secondary genre of a role-playing game, allowing players to progress and unlock more movement abilities and upgrade already existing ones to navigate the game at faster rates. Applying restrictions to the upgrades players can apply at one time will force them to strategically use their available abilities wisely and changing their style of play based upon the current scenario. For example, puzzle-based sections may better be played using a set of upgrades (typically called a build) that benefits movement and its control. On the other hand, while in combat it could be better to use a build that increases one’s health and damage dealt.

## Computational Methods

|  |  |  |
| --- | --- | --- |
| Method | Why | How |
| Thinking abstractly | Referring to the process of separating ideas from reality, it can be understood in two separate ways: abstraction for the developer, and abstraction for the player. For the developer, abstraction refers to the difference between the code and the hardware, and hence creating a model that will make the best use of the hardware. For the player, abstraction is the difference between the information given to them (such as the GUI, colours of effects) and the actual code and logic behind it all. | Only the bare minimum information is provided to the player, making sure that they only know what the developer wants them to know. Workings of certain mechanics could be summarised within a paragraph or shown through cutscene tutorials. Information on weapon states can be shown with images instead of text. As much as possible should follow the show-not-tell principle and detail does not need to reach that of realism. |
| Thinking procedurally | Otherwise known as decomposition, this is the process of breaking a large problem into sub-problems, making the overall task much simpler. Without the ability to decompose, designing and creating a game would be completely impossible. People already decompose problems to solve them. By splitting a huge problem into extremely simple ones, it makes the entire task far easier and reduces the time wasted in trying to solve a complicated problem. All the saved time will create a better product in the end. | The task of creating a game can be split into the many parts of game development, such as programming, art design, and other things. Within programming, it can be used to split the game into separate modules, classes, and functions. Grouping together related functions with importable modules will reduce the amount of code one needs to write and is better to maintain and debug. It is also far easier to create a function that does only one job and have a separate function call all the pieces when they are required, avoiding code duplication and errors that might occur when doing so. |
| Thinking ahead | When designing programs, rewriting the same code or similar code over and over again can be boring, inefficient, and a pain to debug, while also reducing the readability and increasing how error-prone it becomes. It also becomes far faster to program if all the steps have been planned out beforehand, such as inputs and outputs. Identifying spots where one can cache data to improve data access speeds and reducing memory usage will also allow the game to run faster, while leaving more resources for other parts of the program. Thinking ahead also references planning out the timing of when to implement features and how long it should take to do so. | One can simplify repeatable code by extracting them into functions and their respective calls. Taking this a level up means using classes (and hence Object Oriented Programming) to reduce code duplication as much as possible. This fits in with using Unity and C# since it uses objects as the framework, and so programming using OOP will be far easier and better supported. When multiple objects may access the same data, this data will be cached in a static section of the class, increasing memory efficiency. Techniques like polymorphism and abstract classes will further reduce the need to rewrite code. |
| Performance modelling | In games where skilfully timing and predicting trajectories is the aim, having any source of freezing or lag (when there is a significant difference between the player’s inputs and the outputs displayed on the screen) will result in a sub-par, even rage-inducing experience, due to issues out of the player’s control. While this is the intention for some games, it is not the intention for mine, and as such, code should be written efficiently to improve the player’s experience. Performance modelling will help in analysing inefficiencies and hence places that could be rewritten, resulting in an overall all better game. | While doing the programming can be more enjoyable than other parts of the development process, wasting time testing inefficient and longwinded solutions can be a massive bane that may end up costing you the project. Being able to evaluate the performance of a function or script before it is rigorously tested by modelling it and quantitative analysis will help identify weak points in algorithms and potential places to improve the efficiency, reducing the lag a player might experience. One useful tool to quickly measure a function’s efficiency is Big O Notation, showing how effective the algorithm is as the input size increases. |
| Thinking concurrently | Not every process in a game can run sequentially. Sometimes things must run simultaneously to run properly, such as physics calculations alongside GUI rendering, and especially input sensing, which must be handled concurrently with other inputs, such as moving and firing. Also, waiting for other tasks to finish or acquire data can block the CPU, which is more wasted time, so allowing the CPU to switch to another task while it waits is efficient use of its cycles. | The Unity Engine already manages a multitude of processes concurrently by assigning them to separate threads, which can then be picked up by separate cores. Other tasks such as highly parallelisable numerical tasks and graphical tasks can be offloaded to the GPU, which performs the same code on many different data (SIMD architecture) in a huge array of parallel cores. |
| Thinking logically | Without the ability for the program to change what it performs based upon the inputs, many projects and pieces of software would be virtually impossible to create. The ability to loop code also significantly reduces repetition of code, reducing the error rates from poorly duplicated code, and making it easier to debug. | Making different functions run when different input keys are pressed is the entire backbone of games, and as such, having the ability to change control flow is incredibly important. Not only that, but having code run on certain conditions being fulfilled makes sure that only necessary code is run, and avoids potential crashes. For example, if a raycast (shooting an imaginary ray and checking if it hit any object) hits an object, the pickup animation might be different for different weapons if the object hit was even a weapon in the first place. In certain situations however, using a switch statement is more efficient that using many if statements, since in a compiled language, the compiler simplifies switch statements into a switch table, which is a table of all the relevant memory locations for the different parts of code, and a hashing function is used to map the match to the memory location. This ability is more efficient than evaluating many if statements. Just as integral is the ability to have all the game code, or at least the relevant sections, be rerun at the start of the next frame, allowing for an interactive experience and to make sure that the game can be actually be played. Looping can be used elsewhere within movement calculations and predictions or for iterating through lists and arrays. |
| Backtracking | When programming, it is not always sure beforehand whether a solution will work, or if is worth implementing a certain way. As such, you may have to backtrack to previous working solutions and try a different approach to settle upon a far better solution. Over time, this maintains or improves the code, will also keeping the development enjoyable, and motivation high. | Sometimes, potential optimisations may be more hassle than the old solution and one’s time could be better spent elsewhere. If at the time such a solution does not present itself, you can leave it for the time being and come back to it at a later date, when your brain as refreshed and other parts of the code may be completed. This allows you to come up with fixes that you may not have thought of before, since you can leverage the other code, and a fresh mindset. |

## Stakeholders

A stakeholder is a person/organisation that is interested in a project, in this case, my game. These people have the power to change significant parts of the documents as they are the ones who the developers must consult when implementing certain features and how those features will work. For example, these people might be users of the program, such as the players of a game, or they could be producers and publishers wishing to make money from selling the program.

### Silly Australian ® - Henry Masters (Founder and CEO)

My main stakeholder is the founder and CEO of the amazing game development company, Silly Australian ®, Henry Masters. Masters is a huge fan of precision platformers with lots of challenge, as they prioritise a skilful experience that one can get better at, constantly keeping them within the flow state. Some examples are Celeste and Super Meat Boy, both being loved by players of the precision platformer genre. However, they also like role-playing games (RPGs), such as the hit 2015 indie game, Undertale from the brilliant mind of Toby Fox, and The Legend of Zelda: Breath of the Wild from well-known console and game production company, Nintendo Co., Ltd.

Recently, the company is wishing to start adding a secondary part to their business by publishing other small games under big names, and help with the marketing of the experience to allow the developer to reach a larger audience than they may have gotten without going through the publisher. As such, they are willing to provide feedback on the actual code of the game, making sure it is optimised for various devices which may or may not have the greatest hardware, preferring efficient solutions as well. This leads to them requiring a game that runs fast and smoothly as their top priority, since the game should be accessible to as many people as possible, and they are willing to have compromises on the visual clarity of the game and the background story of the game as this does not contribute to the gameplay experience.

### DrunkDriving ™ - Amarveer Flora (CEO)

My secondary stakeholder, handling the playtesting of the game and complexity of the playstyle is Amarveer Flora, the CEO of DrunkDriving ™. Also a fan of difficult combat in games such as Elden Ring and Dark Souls, they have experience with massively multiplayer online role playing games (MMORPG) such as Hypixel Skyblock and Black Desert Online. Similarly to these games, they also like the simple but logical gameplay that comes with automation games like Beltmatic and the Minecraft modpack, Techopolis 2. Importantly, they have great experience with shooter games like Overwatch 2, Fortnite Battle, and Call of Duty, while having great reaction times. This makes them a great stakeholder to test the game for bugs while being able to play at the recommended difficulty of the game. They similarly prioritise the player’s gameplay experience, but they are more focused on the bugs and the instability of the game. Having many bugs in common gameplay will lead to an unfair and dissatisfactory play, so their priority is fixing as many of the bugs as possible, such that a player does not ever experience a frustrating playthrough.

## Research

### Titanfall 2

Released on 28th October 2016 by Respawn Entertainment and published by Electronic Arts, Titanfall 2 is movement shooter, where the player swaps between moving around as a human character called a pilot and playing within a Titan, where the style of play completely changes. As a human, the player has access to wallrunnning, sliding, double jumping, and other abilities depending on the level. As a Titan, the player is able to gain great combat abilities, specific to the kit that they choose to apply to the Titan, all of which overpowers a human in every aspect. However, the movement capabilities of the Titan are reduced, leading to the only shared ability between all Titan kits to be the ability to move around. Some kits can fly, others can dash, but most of them have no extra movement capabilities. The 20-foot-tall war machines also struggle to get in some places, and as such the player will need to frequently exit them to reach other places or remove large volumes of health from enemy Titans via hijacking. This excellently illustrates the design philosophy for each of the levels in Titanfall 2 – ‘211’: two parts pilot combat, one part pilot movement/puzzle solving, and one part Titan combat.

When designing levels, the team at Respawn Entertainment made each of them based upon on a different feature. For example, in the level Cause and Effect, the player swaps between the past and the present, managing enemies in both timelines, and using the changes between the two to help in accessing different areas. This planning technique is something that would be wise to follow, but since levels are designed to be completed in speed (and making long levels can be difficult) it might instead be smarter to split a theme across multiple levels, which are typically called zones. It should not be overdone however, as the feeling of ‘wanting more’ is always better than the feeling of boredom.

### DOOM (2016) and DOOM Eternal

Orthogonal Unit Differentiation

Distinctive enemy design implying attack

Glory kill system

Aiming to restart the fantastic and beloved series, id Software created DOOM and released it on May 13th, 2016, 12 years after the last game in the franchise.

### Half Life (and other Source Engine games)

Half Life was the inspiration for many of Titanfall 2’s puzzle design, where there is a minimum amount of momentum required to traverse and solve the puzzles. Released in 1998 as Valve Corporation’s first game after 2 years of work, Half Life set the standard for what an amazing story-based first-person shooter game should be. It carefully and cleverly introduced the player to all its mechanics, and showing the depth that they could be used. Across the 19 chapters, and approximately 12 hours of immersive gameplay, Valve crafted an experience that tells the story of the creation of a dystopia, and Gordan Freeman (a young scientist) along with his attempts to stop it.

Valve built their debut game on GoldSrc – an extremely modified version of the Quake Engine and the Quake II Engine from id Software (the developers estimate that the resulting code is over 75% their own). The Quake Engine had a feature within its movement calculation code called ‘air acceleration’, which as its name suggests, allows the player to accelerate in midair. Normally, this wouldn’t matter, since the drag from moving forward and the deceleration that you experience when touching the floor is enough to cancel all speed increase. However, with a special movement technique called air strafing and bunnyhopping, it’s possible to gain speed, essentially to as fast as you want. Valve later modified and upgraded this to make the Source Engine which debuted alongside Half Life 2 and Counter-Strike: Source, and later on the Source 2 Engine with Half Life: Alyx. Since all the later engines were based of the previous engines, they all contain the same similar movement code and hence speedrunning techniques, which is incredibly useful for speedrunners since they can reuse their own skills.

The Half Life franchise is also well known for its incredibly realistic AI, which is especially well demonstrated with the Overwatch Soldiers from Half Life 2 and the following games in the series. The soldiers are capable of replicating and enforcing blitzkrieg-style military tactics against the player and making smart decisions in the usage of their weapons. There is also a clear hierarchy present within each team, and the behaviours of the different soldiers are cleverly explained behind the story of the game.

The Combine AI functions as a multi-part system, with each enemy following a set of logical steps, then reporting back to a manager system, which assigns the respective enemies the appropriate roles based on the number of targets. Each soldier will then maintain that slot until their schedule finishes, which is a set of tasks assigned to it, such as finding cover, or firing at the player. This complex system allows for the tactics that the enemies should enact, while also keeping the situation manageable for the player, since the system will not allow more than 2 enemies to attack the same target simultaneously, which makes it a brilliant basis to design the enemy AI for my game as well.

### Ultrakill

Movement + weapon interaction/combo

### Omori

An absolutely phenomenal role-playing game (RPG) by OMOCAT using RPG Maker MV, Omori tells two interconnected stories of two connected teenage boys, dealing with both friendship and guilt. The bulk of the game is spent interacting with characters in both the storylines and completing quests for all of them. Dialogue systems in games can be incredibly complex and important, giving one an insight into how the characters of the game are feeling, and the possible options can change based on what the player has done. If the player has completed a quest, the system must be able to detect the change to the relative variables and change accordingly.

The importance of dialogue in a game comes not from the words, but the feelings they invoke and pass on to the player. After all, if the only things a character says is just describing the current situation, better world design or even a signpost is what you need to have ‘dialogue’, instead of implementing such a complicated system. However, the inverse is not true – a signpost can not represent emotion as well as a set of characters, no matter whether they display the emotion or not.

Omori’s system links together facial emotions and text effects to further accentuate the feelings of the characters. I, on the other hand, do not need to invoke emotions through facial expressions, nor do I need such extravagant text effects such as shaking and waving. However, applying colour, italicising and bolding text is useful in pointing out the most important things to the player while they are skimming through the text, while also helping them retain those specific pieces of information. In a game with a lot of stress, being able to skim dialogue quickly is important.

### Conclusion

### Interview

## Essential Features

### Movement

#### Walking/Strafing

Walking is the basic movement action that the player can enact and will need this to start of many other actions. Comparatively to other ground-based movement, the speed of this will be situated in the middle of the rest.

#### Sprinting

While walking can be all one might need, getting to places faster is important to avoid getting hit in combat, or complete certain puzzles within the required timeframe. Sprinting will increase the ground acceleration that is enacted, and the player will also have a higher top speed compared to walking.

#### Jumping

Another staple of the movement actions, jumping allows the player to easily gain vertical momentum, and it can be developed by allowing the player to jump while in the air, or jumping off walls.

#### Crouching (both toggle and hold)

Crouching will shrink the player’s hitbox vertically, granting access to smaller passageways, and providing a potential advantage in combat, since the area that enemies can attack the player in is far smaller. To balance this out, crouching will force the player to move much more slowly and will be the slowest ground-based movement option.

#### Sliding

When above a certain speed, using the crouch button will instead lock the player into a sliding state. Sliding should first increase, then decrease the player’s speed, encouraging you to leave your slide earlier to gain speed. The target direction will be the current direction of movement, but input will only allow slight nudges in direction, rather than a controlling factor, allowing the bare minimum control. Friction while in a sliding state will also be a lot lower than other states.

#### Wallrunning

A much more advanced movement option, this makes the player stick to walls and allow them to move along it. This opens the possibilities of combat terrains and puzzles and is usually included in movement-shooter games. The player can also jump off these walls, which may lead to a variety of solutions to different situations the player may encounter.

#### Grappling

While not as common in other games, grappling is still fun to use, but is also far harder to control, since this requires one to plan trajectories. It is also much harder to implement but is still well enjoyed by many players in the game’s it shows up in, such as the multiplayer mode of Titanfall 2, or as a basic movement option of Ghostrunner.

#### Double Jump

This is a far more common ability, where while in air, the player can press the jump button again to gain an extra boost vertically, akin to a jump on the ground. Without this, the player is only permitted to jump while touching the ground.

#### Boosting

Like a dash that might appear in other games, a boost is a multiplicative increase to the player’s speed and will allow them to gain speed much faster than attainable through other methods. Due to the potential of exponential speed, both a maximum limit for boosting must be set, while also implementing a heavy cooldown for using it.

### Combat

#### Fire

On the combat side, firing is an essential action. Without this, it would not be called a movement-shooter. Firing will be a catch-all term for using weaponry, such as slicing with a blade, shooting a weapon, or throwing a grenade.

#### Zoom

While a player may not use this as much when they are moving around a lot, it is still a staple of the shooter genre to include some sort of zoom capability on weaponry or on your visuals to be able to see long distance clearer. This may have a side effect of encouraging a stealthy and cautious playstyle instead, but this could be negated by having more difficult sections later in the game.

#### Swap weapon

From the beginning of first-person shooters, the usefulness of having multiple weapons and the strategy that appears from doing so was always an incredible lure into the games of the past. This inherent feature stuck through the generations and remains as an expected feature in a shooter. However, starting from Halo, the number of weapons one could carry could be limited, due to the limitations of the controller at the time. As such, another trend of having a limited number of weapons that the player could equip at one time rose in popularity. It’s an incredibly easy way of bringing depth into any shooter, by making the player have to choose which weapons, they want to use, and by having a limited ammo capacity, also making the player choose if they can wait to get ammo, or if they need to change weapons now.

#### Quick melee

While unlikely that the player might repeatedly get within close range of any of the enemies, there must be a close, high damage option for the player to enact. The melee is a powerful move, that depending on game-to-game, can be one of the highest damage options available. This ability allows the player to use any weapon to inflict a strike onto an enemy, no matter the usual use of the weapon.

#### Pick up weapon

Since the player can run out of ammo, it is likely that they will end up without any weapons with ammo in them, other than blades. As such, it is possible for the player to pick up weapons dropped by enemies or placed in boxes throughout the level to counteract this difficulty.

#### Reload

Guns do not have a limitless capacity of ammunition, and to help players to immerse better with the game, the player can reload the magazine in the currently active weapon if applicable to do so. This can add a further layer of strategy to the game, as a player can decide to reload in the middle of a magazine before entering a gunfight, or not do so if they fear they will get hit in a moment.

### Sound Effects

The brain is very good at linking different senses together for the same piece of information. Having different sound effects (sfx) link to different actions will act as either a confirmation that the action has been processed, or it may warn the player of danger.

### Saving

Saving is needed for longer games, since players can get incredibly frustrated with losing progress every single time they close the game. As such, saving has been implemented in many major games as a basic feature. Different games do the system slightly differently, usually splitting into 3 different types. The first is where the player has many slots and can save the game into any of these save slots and have multiple different places they can pick up from. This can also be used to have entirely different playthroughs existing at the same time. The second method has only one save slot but saves progressively by making a new save every single time. This only allows for one playthrough, but many checkpoints that the player can load back into. The third method is the one that I shall implement, where a mix of the other two types is used. Specifically, the player will have multiple slots, where each slot denotes a playthrough. Each slot will have multiple versions based on manual saves or automatic ones based on checkpoint.

### Multiple Levels

Having many different levels will increase the playtime of the game and introduce variety, so that the player doesn’t get bored with repeating the same level with no differences. These levels will be split into zones/chapters where levels in the same zones/chapters will have the same thematic style and mechanics.

### Glitch

The glitch meter is a measure of how aligned to the default programming the player is. Pacifism goes against the original programming: this decreases the glitch meter. Killing follows the original programming: this increases glitch meter. Secrets decrease the glitch meter, as they reveal the true nature of things. A lower glitch meter would represent the player character questioning the morality of what they have always been taught and programmed to do. It will cause:

* Decrease in health regeneration speed.
* Increase in damage taken.
* Player will see hallucinations.
  + Grants the ability to attack friendlies.
  + Enemy civilians may be on your side.
  + Characters in names and signs may move around.
  + Relates to PTSD type event at the beginning of the game – the only cutscene of the game.

A higher glitch meter represents the player character believes what they have been taught and programmed is the right thing, leading to:

* Increase in health regeneration speed.
* Decrease in damage taken.

This works with the moral of the game - doing the right thing is almost always harder than doing the wrong thing, and sometimes prejudice will perpetuate persecution.

### Scoring System

Each level is scored based on:

* Time to complete.
* Enemies killed.
* Secrets collected.
* Inverse of death count.

Further point bonuses will be granted for:

* Using less weapons (melee only)
* No deaths
* Pacifist (only applied on the final level if the player has killed zero enemies across the whole save)

This should encourage players to return to the level and get a higher score, potentially seeing the flower planting system. The score will be displayed in a room, and the player can interact with each of the elements to show the score for each category.

### Merit

The currency of the game is called merit. It can be obtained by picking it up from within the level (such as the bodies of friendly soldiers) and from the score at the end of the level. Killing increasingly difficult enemies will lead to more merit, although to stop merit-farming tactics, it will be limited to this equation:

.

Merit will allow the player to buy upgrades for their player character and unlock more weaponry/ammo types to use in the pre-level inventory setup. However, ability unlocks will be based upon the level progress, since it will be tied to the story.

### Dialogue System

While it should not be used extensively as it may break the flow of the game, this system should be complex enough to handle all dialogue requirements it needs.

It must:

* Allow for one to easily add multiple languages
* Allow for timing cues
* Allow for branching options
* Allow for quest completions
* Allow for different dialogue based on external events
* Allowed to change variables
* Prompting characters to face different characters
* Allow for changing text colour (this should allow for accessibility changes)

### Death Screens

To encourage the player to use healing items and use better movement to avoid taking damage, too much damage will result in the player’s death, and a screen will need to pop up allowing the player to load the most recent save, and potentially showing a useful tip on the screen to avoid death the next time. The screen could also give other statistics such as how many the enemies had killed.

### Graphical User Interface (GUI)

Many users are not exceptional at navigating and using command-line interfaces, and the human brain can process images faster than it can process words, and using the mouse is more accessible than using text arguments. As such, I will be implementing both a GUI for menu navigation, and a Heads-up Display (HUD), which will communicate information to the player while playing the game. Some elements of the HUD may be in text and numbers, while other will be in graphical solutions such as bars and images.

## Limitations

Within every project, there are limitations that one must consider, since not everything that one would like to do can be accomplished. Sometimes, trying to implement everything you want will in fact ruin the final project. Instead of trying to break all the constraints, one must instead make considerations and work around these considerations. If possible, some of the limitations could be considered challenges and you should work to break it down and produce a better product, but the costs of doing so must be accurately estimated and the reasons must be weighted accurately against the problems it may cause.

|  |  |
| --- | --- |
| Limitation | Why? |
| Time | Time is the most obvious limitation – the never-changing constraint that will limit how far I can progress with the game. With infinite time, everything in the game could get infinitely refined, and many features could be added to the game. However, infinite time is not something we have the luxury to. In fact, the entire project must be done within the space of 8 months. Therefore, there will be features that would be desirable but can’t be implemented and some features which will not be as refined as preferable. |
| Budget | The budget for this entire project is nothing. This is a severe limitation meaning that nothing can be outsourced, and while usually you could hire better programmer, and an entire sound design team, nothing of the sorts can be done here. This will have repercussions for the software limitation as well and means that if there are any textures or models I want to use, they must be free or made by myself. |
| Hardware | Maintaining the performance of the game is very important because not everyone has access to top-notch hardware, and the game must run extremely well since speedrunners wish to have zero-latency from sending their actions to receiving feedback on the screen. Therefore, the game must be made as simply as possible, with as much optimisation as well. However, I am also unable to test on different types of hardware, and so the best I can do is testing on Windows on my own system. Furthermore, the game will be only for the PC, since I do not own a console, nor can I get the game files onto a console if I owned one. |
| Software | Software is also a constraint since I can’t acquire all the professional applications, and some applications I can’t run due to my own hardware. Since the budget is zero, all software and third-party resources must all be free, or hand made. |
| Proficiency | While I have used Unity and C# in the past, they were all for basic projects that were copied from tutorials, and as such I will need to learn more of the quirks of the engine and its language, which will take even more of my time to build the game. |

## Solution Requirements

While it would be ideal to have all of the essential features included in the game, not every single one needs to be implemented to the same level of usability as others. Based on the limitations, some features may need to be cut from the game, as is typical within the industry.

|  |  |
| --- | --- |
| Requirement | Justification |
| Movement and Combat | Both movement and combat are single-handedly the most important requirements to exist for this NEA, since they define the primary genre of the game. Without either of these features, it would be incorrect to call the game a movement-shooter. |
| Saving | Every single modern game that is released nowadays has saving since no player is ever expected to finish an entire game in one sitting. To maintain the player’s retention, they must be able to return to most of their progress still available to access. |
| Multiple Levels | For such a fast-paced experience, the game would be unsatisfying and not an adequate experience if not for multiple levels. This will extend the playtime of the game and allow the player to better integrate with and learn the mechanics of the game better. |
| GUI | A GUI is integral for the user to be able to interact with the game, since many engines do not allow the player direct access to a console/terminal for a command-line-interface. A CLI is also far more difficult to use and is less accessible to the average gamer, making this a requirement of the final project solution. |
| Death Screen | This also includes the player being able to die, and this is incredibly important for gameplay. Death is the risk factor, and it is what the player is intending to avoid as they are rushing around and firing at the other enemies. With no risk, there is no need for the player to attempt to play the game as intended. Once death has been achieved, it is important that the player is able to go back to their previous save or restart the level to continue playing. Any hints on how to avoid death are an optional extra to add to the death screen. |

### Hardware and Software Requirements

|  |  |
| --- | --- |
| Requirement | Justification |
| Computer Hardware | The game is intended to run quickly, with as minimal delay/lag as possible, therefore the GPU requirement is only the need for integrated graphics, or a simple graphics card such that one can have an enjoyable experience, while the CPU requirement is for a chip that can adequately support its GPU. On the side of system memory, the game will require a minimum of 2GB of RAM, so the system would be recommended to have 8GB of RAM, so as not to cause disk thrashing with other programs that the user may have loaded in, which only less that 2.5% of people who play games do not have access to nowadays. |
| Peripherals | The game will have support for both a mouse and keyboard combination (kb/m), or a controller as an input device, but nothing else, as it would be difficult to port all of the controls that would be needed to play the game to another appropriate input system, such as touch controls. The kb/m and controller input schemes are also the most commonly used ones, as evidenced by all the games in my research having support for both of them. |
| Operating System | My project will only be available upon a 64-bit Windows system since that is the most widely used type of system for playing games, coming in at 96.78% of all gamers on the Steam platform according to the Steam Hardware Survey of August 2024. |
| Storage | One would need as much storage as is required to contain all the files of the game and have at least a few gigabytes leftover for a swapfile/pagefile, in case the computer needs to use virtual storage. |
| Software | Many games use the same set of C++ Redistributables and depending on the individual’s proficiency in playing games, they may or may not already have the relevant packages installed. This also includes .Net 4.0, the appropriate drivers for their device and the XNA Framework Redistributable. Furthermore, if the user wishes to use a controller, they will need the right software to connect the controller to their computer, which is usually Microsoft’s XInput, but may also be DInput (an older version for support for more buttons). The appropriate software and drivers are dependent on the user’s own controller. |

## Success Criteria

|  |  |
| --- | --- |
| Criterion | Justification |
| Strafing Movement | Strafing is the barebones required option for moving, allowing access to 2 dimensions, and even the third dimension when slopes are introduced. |
| Jump | While strafing can allow the player to move vertically, it is dependent on the level geometry, and it would instead be a better option to allow the player the ability to jump while on the ground, and maybe even a double jump while in midair. |
| Crouching | To avoid the enemy gunfire or to enter certain sections of puzzles in the game, crouching is a criterion as it allows the player to decrease the surface area that is exposed. |
| Firing weapons | As mentioned, the player must be able to fire weapons to retaliate and survive for longer against their enemies. It is also part of the genre’s title and would be fraud to use the genre ‘movement-shooter’ without the ability to shoot. |
| Can take damage | While the player can damage others, in turn, they too must be able to take damage, allowing an even playing field and forcing the player to take better and safer movement routes, or prioritise more dangerous enemies. |
| Ability to go back to a save on death | Once the player’s health reaches below 0, the only logical option is for the player to die, which reinforces the risk with taking damage. The player must then be able to continue playing which in a singleplayer campaign, is usually done by reloading a previous save. The death screen will contain a button to do so, by selecting between a variety of saves. |
| Saving | This allows the player to retain their current progress for later, such as when they close the game, or if they suspect that they might be in a tricky situation. |
| Loading | Conversely, one must be able to load the save that they made to bring meaning to the saving feature and re-enter the state that they were in when making the save. |
| Menu with buttons | The GUI is the most accessible method of navigating through menus and interacting with the game. Out of all of these, the button is the most intuitive and useful of selecting options, instead of a scrollable menu. |

# Design

## Decomposition

### Combat

#### Weaponry

##### Swapping

Being able to swap out your weaponry on the fly is what encourages players to try new things in the game. This swaps the currently selected weapon with the weapon that the player is looking at.

##### Guns

1 of the 3 types of weaponry is guns that allow the player to deal damage from range, firing bullets that damage upon clicking the button via a raycast.

##### Melee

The second type is melee weaponry which deal high damage from close range, like bats and blades, requiring a collision check instead.

##### Grenades

The last type is grenades, which are lobbed, following a physical arc and dealing explosive damage based on its range.

#### Player

##### Movement

###### Horizontal

All movement that uses the basic strafing keys are handled similarly within the same function, moving the player in the direction of the keys relative to the current facing angle in the horizontal plane. This will all be held in one function, that references other functions to access the relevant acceleration values for the different movement types and the speed limits.

###### Vertical

This encompasses jumping, gravity and the lack of control the player should have when moving in air. While the player is not on the ground, gravity will constantly accelerate them downwards. Jumping allows one to get over an obstacle by adding vertical velocity, but it is also possible to ramp off slopes and get far more verticality that way. Due to this, one will have lower control of the character once in the air, needing them to plan their original trajectory. Ramping will be automatically handled by the physics system that is built into Unity 3D, while both jumping and gravity will be separate functions. Jumping will be triggered upon the input being pressed, while gravity will be triggered every frame

###### Wallrunning

Wallrunning is a special case of movement since the plane of movement then becomes that of the wall you are running on, and the player can only move forward on this wall, fall off or jump off. This allows them to stick to the wall and is an entirely separate movement option than the others.

#### Entities

All entity types are shared between allies (friendlies) and enemies, bar the civilians which are only on the enemy side, as is in line with the setting of the game.

##### Drones

An enemy that flies around and maintains a height-advantage on the player where possible. Depending on their type, they may act differently, such as shooting from afar for ranged drones, or moving towards the player for explosive and melee drones.

##### Soldiers

An enemy that moves around only on the ground and will drop weapons and ammo when killed. Will stay away from the player but will attempt to maintain line of sight.

##### Tanks

Large and slow ground-based vehicles that deal heavy explosive damage occasionally. Have exceptional range compared to any other type of troop.

##### Civilians

Cannot deal damage and will run away from enemies.

### Environment

#### Ground

Allows for all basic movement and disables gravity while the player is stood upon it. It also prevents the player from falling out of the bounds of the level. Moving up a slope will convert some of your horizontal velocity to vertical velocity as per the physics dictate.

#### Walls

The player can wallrun upon them but loses all other types of movement except jumping. Also disables gravity and prevents access to certain places.

#### Pickups

Items will be scattered around the level, such as coins or weapons that the player can pick up and use, encouraging exploration and challenge.

#### Interacts

Objectives for the level can be anything and are interacted with. Other things like doors and upgrades can also be interacted with.

### UI

#### HUD

The heads-up display shows important information to the player, such as their health, what the active weapon is, how much ammo that weapon has, and outlines for enemies for clarity. It cannot be interacted with but is a necessary part of gameplay.

#### Menu

##### Title screen

Loads on startup, allowing access to the options menu and save menu, along with exiting the game. Displays the name of the game

##### Options

Changes settings such as game volume and mouse sensitivity.

##### Pause

Pauses the game and allows the player to return to the main menu. Does not save the current state of the game.

##### Pre-level inventory

Provides a GUI for the player to assign upgrades appropriately and select the contents of their weapon slots.

##### Save menu

Displays all the save slots, each displaying 4 things: the slot number, the save history menu, the time spent in the save, and the name of the level.

##### Checkpoints

Will automatically save the progress of the player in the game at these points, creating an autosave.

###### Load game

Once a save has been selected, it will load the attributes into memory and load the level

### Dialogue

#### Interpreter

Reads the .dlg file and translates the instructions to instructions/lines for the manager to assign.

#### Actors

Will look towards appropriate actors according to the interaction and message sent by the manager.

#### Managers

Handles the receiving of instructions from the interpreter and assigning the appropriate actors their roles, or sending the subtitles/choices to the subtitles text box

## Structure Definition

## Algorithms

procedure SetPlayerDimensions(height, radius)

previousHeight = playerCollider.height

playerCollider.height = height

player.transform.position = player.transform.position - Vector3(0, (previousHeight - height)/2, 0)

endprocedure

This changes the dimensions of the player which will be shaped like a capsule (a cylinder with hemispheres at either end) to be set to the parameters of height and radius, allowing for the scaling of the player character and crouching, then moving the player character down the difference in height.

function CrouchControlState()

holdCrouch = playerInputs.Player.HoldCrouch.inProgress

if playerInputs.Player.toggleCrouch.triggered then

if toggleCrouch == true then

toggleCrouch = false

else

toggleCrouch = true

endif

endif

if holdCrouch == true then

lastHoldCrouchState = true

return true

else

if lastHoldCrouchState == true then

toggleCrouch = false

endif

lastHoldCrouchState = false

return toggleCrouch

endif

endfunction

Since the player can crouch using two separate keys, this function decides what the appropriate crouch state is, based upon the values of the keys pressed and their previous values.

procedure Gravity()

if NOT isGrounded then

movement = movement – gravity \* Time.deltaTime

endif

endprocedure

So long as the player does not have contact with the ground, they feel a constant acceleration of gravity downwards.

procedure Crouch()

if lastMovementState != movementState then

if movementState == PlayerMovementState.crouching OR movementState == PlayerMovementState.sliding then

SetPlayerDimensions(crouchingHeight, playerRadius)

else

SetPlayerDimensions(standingHeight, playerRadius)

endif

endif

endprocedure

When the player has just changed their movement state, and that state is from standing to crouching, or vice versa, it calls the relevant function with the correct dimensions.

procedure Movement()

acceleration = baseMovementAcceleration

target = player.rotation \* inputDirection

alignment = Vector3.Dot(groundVelocity.normalized, target.normalized)

if groundVelocity.magnitude > maxSpeed then

acceleration = acceleration \* groundVelocity.magnitude / maxSpeed

endif

direction = target \* maxSpeed - groundVelocity

direction = Vector3.ProjectOnPlane(direction, groundNormal)

direction = direction.normalized \* acceleration

direction = direction - direction \* frictionMultiplier

movement = movement + direction \* Time.deltaTime

endprocedure

This is the main movement code, converting the player’s rotation and inputs into a movement in that direction, adding the movement to a global movement variable.

## Usability

### Game Design

#### Main Menu:

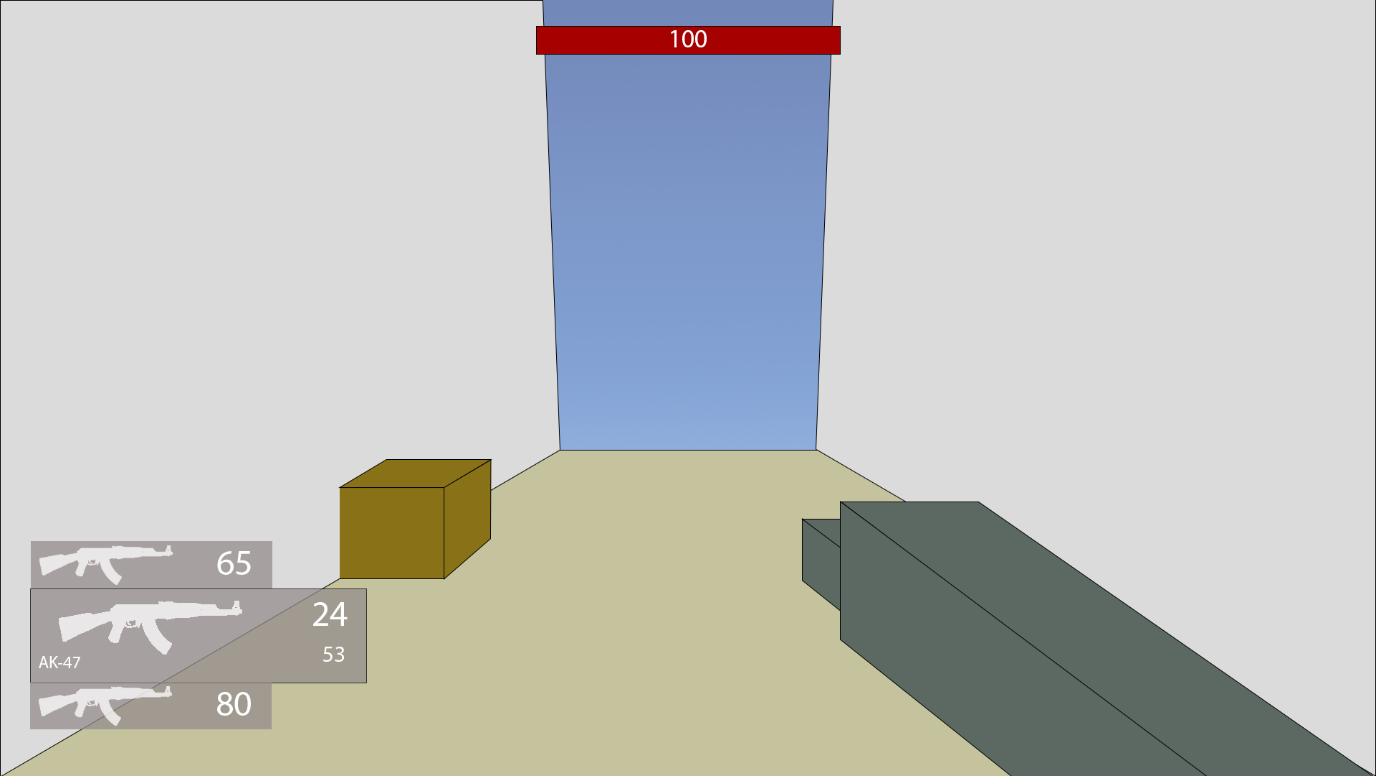


#### Save Slot Menu:

A screenshot of a game

Description automatically generated

#### Game HUD:



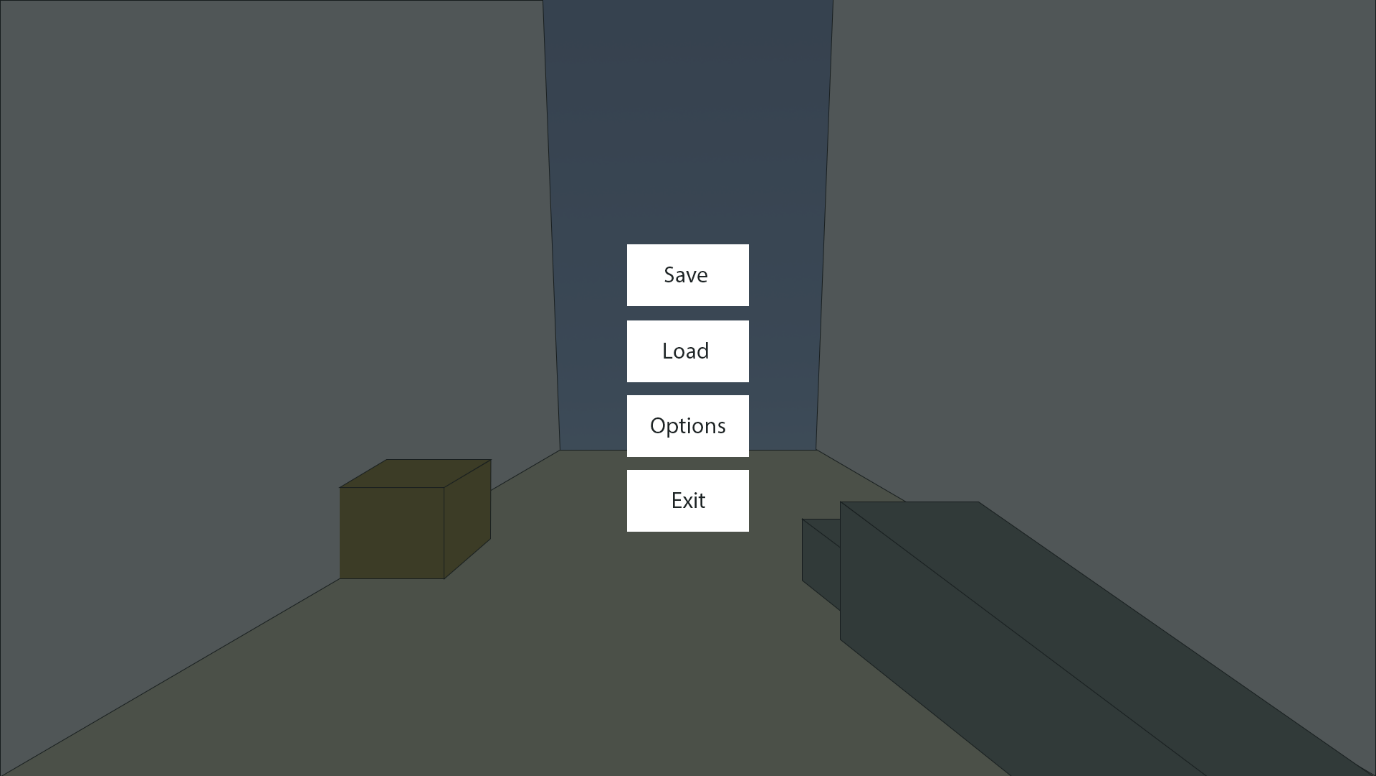
#### Choice:



#### Subtitles:



#### Pause Menu:



## Variables and Data Structures

|  |  |  |
| --- | --- | --- |
| Type | Name | Justification |
| Movement | | |
| Float | maxSpeed | Limits the player’s speed so that they don’t go too fast and break the game |
| Vector3 | movement | Contains the total movement of all the different movement functions, which is then applied together at once |
| Float | standingHeight | The height of the player when standing |
| Float | crouchingHeight | The height of the player when crouched |
| Bool | isGrounded | Used by the gravity and jump functions to check if the player is touching the ground |
| Float | maxSlope | Used by the collision detection function to partition collisions into floors and walls |
| Float | gravity | How fast the player should accelerate downwards |
| Float | jumpForce | The speed the player leaves the floor when jumping |
| Int | airJumpsTotal | An upgradeable value that makes sure that combined with airJumpLeft, controls the ability to jump in midair |
| Int | airJumpsLeft | Makes sure that the player can’t jump away too far |
| PlayerMovementState | movementState | The current state the player is in such as walking or crouching |
| PlayerMovementState | lastMovementState | Used by the crouching function to check whether the player has changed to crouching from a previous state to then change the player’s dimensions |
| Float | friction | A percentage of the player’s speed that is taken away per frame while on the ground |
| Float | drag | Same as friction but for in the air |
| Bool | holdCrouch | Stores the state of the key responsible for the crouch that must be held |
| Bool | toggleCrouch | Stores the state of the key that can toggle crouching |
| Bool | lastHoldCrouchState | Used to detect a change in the state of holdCrouch |
| Dialogue | | |
| Database | Dialogue | A database is the best way to store similar/linked versions of the same piece of data such as different languages of text and select the right language at the appropriate moment. |
| Save System | | |
| Text File | Save Lookup | This needs to be used to quickly load the last used save on a slot |
| TMP\_Text | scene | A reference to the text object on the GUI that is the scene name of the save file. |
| Dictionary<string, string> | lookup | A usable form of Save Lookup for the code to use |
| List<SaveSlot> | slots | Used in the slot displaying system by the GUI after retrieving a list from the SaveSystem |
| Class | SaveSystem | A static class with functions handling saving the player’s data, loading that data into the game, and listing out the saves for external systems. |
| Class | SaveSlot | A temporary container used for the save slot GUI with basic information about a save slot to minimise memory usage. |
| Entities | | |
| Class | Entity | Handles health and damage |
| Class | PlayerEntity | Inherits Entity, allows the player to improve their maximum health |
| Class | EnemyEntity | Inheriting Entity, it implements AI searching, moving, and attacking, along with communicating with a commander. |
| Class | Drone | Inherits EnemyEntity, modifying functions to move in 3 dimensions instead of one the ground. Has far detection range and high speed but weak |
| Class | Soldier | Inherits EnemyEntity, moves on the ground with low detection range and average movement abilities and health |
| Class | Tank | Inherits EnemyEntity, moves on the ground with extremely far detection range and high health but extremely low speed |
| Weapons | | |
| String | WeaponType | An ID for the weapon, allowing it to be loaded by the save system |
| Int | TotalAmmo | The total number of shots left before more ammo needs to be picked up |
| Int | AmmoInClip | The total number of shots left before the weapon needs to be reloaded |
| Class | Weapon | An abstract class that mandates a method Fire(), a weapon type, the ammo left in a weapons clip, and the total ammo a weapon has |
| Class | Gun | A ranged weapon that fires bullets, dealing damage |
| Class | Blade | A melee weapon using a collider to deal damage |
| Class | Explosive | Deals damage to all entities within its range |

### Enumerators

Enumerators are more human readable way for the states of an object. Instead of having:

if gameState == 0 then

print(“Game has ended”)

else if gameState == 1 then

print(“Game is running”)

. . .

You could instead have an enumerator which names the states and makes it easier to read and remember. It’s far easier to remember if the state is equal to running, instead of remembering that the state is equal to 5, which is running.

#### PlayerMovementState:

* Idle
* Crouching
* Sliding
* Walking
* Sprinting
* Boosting
* Wallrunning

#### EnemyState:

* None
* Searching
* Combat

## Class Diagrams

### Entities

A screenshot of a computer

Description automatically generated

The Entity class system primarily manages the health and damage of the entity it is attached to. However, *Entity* is an abstract class used only as a framework for the PlayerEntity and EnemyEntity. The only extra functionality the player needs is to upgrade their maximum health, while EnemyEntity adds far more functionality to the base class. For enemies, it also provides the functions for its AI and communication systems. While called EnemyEntity, it will also be used for friendly entities, since they have the same use and intent, with only the range of acceptable targets being changed, such as from the Empire to Varsia (the two countries involved in the war).

The main movement the AI will use will be a PID controller, which is used in industrial applications to bring a robot system back to the target without getting stuck in a cycle. It will also be looking for entities opposing it within a cone, limited by its maximum detection range.

### Weapons

A screenshot of a computer

Description automatically generated

Weapons are how the damage will be dealt to both players and entities in an interactable form. *Weapon* itself is an abstract class that makes sure all following weapons have the correct fields that the save system uses to save their data, while also making sure that weapons have the same public Fire() method that will be used allow scripts that interact with the weapon to use them.

## Iterative Tests

## Post Tests

# Development

## Evidence

### Input Detection

To handle the detection of many different actions from many different input sources, I will be using Unity’s built-in Input System package. Within this, I can set up different input maps, such as ‘Player’, and each of these maps have different actions, such as ‘Fire’, and ‘Jump’. I can easily assign to each of these actions multiple different input types that activate them. Actions are a special version of an event, which is a way for functions to be run when something happens. So, when the user presses one the inputs, the related action emits a signal to all the event’s subscribers (the term for functions listening for the event), which all activate simultaneously. This is how the game senses the inputs.

Since multiple input maps can have actions that are enabled with the same inputs, only one input map must be enabled at any time, and the same instance of the input map must be used across all scripts. To follow this criterion, a script called PlayerManager.cs exists, of which one of its functions is to instantiate a new set of inputs and enable the Player input map the moment the game starts. The reference to this map is public, meaning any class wishing to get the inputs of the user needs only reference this variable. Similarly, if the map needs to be changed, such as one used to handle UI, any script can do so, and it will permeate across all, making sure that no unintended actions occur when the player is intending to do something else.

### Movement

A computer screen with white text

Description automatically generated

A screen shot of a computer

Description automatically generated

### Look

Handling the ability for the player to look by using their mouse movement was not complicated on its own. However, the movement scripts needed changing to account for the player's new ability to face any direction, instead of the cardinal directions, which was fixed as explained above. Looking can be split into 2 parts: looking side-to-side and looking up and down.

### Collision Detection

### Entities (Damage)

### Interacting with weapons

A computer screen shot of text

Description automatically generatedA screen shot of a computer

Description automatically generatedA screen shot of a computer code

Description automatically generated

### Dialogue

Implementing the dialogue system to fulfil all its requirements was incredibly difficult and arduous. To be able to easily add multiple languages, the best solution to this would be to have every single dialogue line in a database, where the different fields are the different languages. All one would need was the record ID and the related language, and the correct line of dialogue would be loaded.

The interactions will be stored in external files with the extension of .dlg, where the flow of lines will be marked with special syntax, of which will be stored in 2 locations: some within the line data in the database, and others within the .dlg file itself. Within the line itself, the syntax is:

* \_ which denotes the text surrounded by them to be in italics
* \* which denotes the text surrounded by them to be in bold
* - which denotes a pause
* < and > surround colour information about the text written in hexadecimal (#FFFFFF) like “<#FFFFFF example> where “example” will be in white.
* \ is the escape character, allowing any of the above characters to be displayed, so long as it is preceded by it

For the .dlg file, the syntax varies more, since rather than styling the text, it instead defines the flow of the lines:

* Dialog sections are marked with [], which is how the dialog will be able to jump and change based on variables. Dialog sections won’t naturally progress to each other unless a -> is used to go to the next section, allowing the last section to continue forever.
* Choices for the player are marked with | at the beginning, and the first choice is used as the title of the choice. These set a local ‘choice’ variable to the number of the choice, which can then by checked and jumped to after. After each choice, a comma is there to indicate another choice is there, except for the last one, signifying it is the last choice.
* Variables will be how the dialog will react to external events. To mark a variable-based line, it will be preceded by @. This will prompt the game to check for that relevant value being set in the save file. If the condition is fulfilled then the dialog will go to the dialog section that is labelled after the -> that succeeds the condition. Otherwise, it will go to the next line. Chaining together many of these will allow an if-else-like behaviour. If a variable is on a line by itself, it can be assigned to a value. Variables can also be assigned values similarly, instead using ’=’ to set the value, and no -> will be required after.
* To request each line of dialog, the line is prefaced with an actor ID, followed by :: then the line ID. The actor IDs are labelled with #Ax and the line IDs are labelled with #Lx, where the x in both is the record ID in the database for each of them. Using this same system but with two actor IDs instead should send a signal for the first actor to look towards the second actor.

As you might have noticed, this would require me to create a custom parser for all of this, along with managers for the subtitles and databases. To start with, I had to learn how to use SQL in C#. The database did not need to be hosted online, and as such SQLite was a perfectly usable solution of the SQL dialects. It was also an advantage to use this since I had some experience with using this in Python before, and was a somewhat well-used solution in Unity projects. However, the drawback is that SQLite was overall the least-used SQL dialect, as many flock to using MySQL instead. However, the many variations in syntax and its main use as a multi-user access database system is not a capability I require. In fact, SQLite is extremely useful in being lightweight, and its main use is for small applications such as this game.

Having decided on a dialect to use, I searched around for how this was implemented within Unity or C#. Unity contains a plugin within it called Mono.Data.Sqlite and combined with System.Data from C#, I could create database connections using the IDbConnection interface and send commands using the IDbCommand interface. I would need two databases, one for storing the entire list of actors, and another for storing the dialogue lines. Having all the actors in a database would make the parser easier to construct, since it would not need to be aware beforehand of which scene and interaction it was in – it could be easily generalised to handle all situations. A separate manager would be responsible for assigning the actors to the actual characters in the scene, but this would be included in the scene data making this a worthwhile option. I first started off by making a way to create the database and table connection, resulting in the below code:

A screen shot of a computer

Description automatically generated

This code makes a reference to the StreamingAssets folder of Unity, which is how developers are able to include files unchanged with the build of their application. Usually when a file is included, it is serialized to make it easier and faster to load into memory. This also means that the format of the file changes, so it may no longer be possible to access it like you may have before. The way around this is StreamingAssets, which is where all the databases and .dlg files will exist. This version of the code does not take into account the different ways that different OS define paths to files, nor does it take into account that multiple different databases exist. To handle the different databases, and the different names for the columns, the function was changed to have two parameters: a string databaseName, and an array of strings that contained the columns. To handle the array, a string had to be made that combined the whole array into 1 string.

A screen shot of a computer code

Description automatically generated

The for loop would iterate over each string in the array, and add it to the already combined total, along with a comma and a space to separate the different fields. The last field does not need a comma after it, so the last 2 characters will be cut off by taking a substring from the start of columnText, to 2 before the end of it.

To then fix the issue of the different OS, instead of concatenating strings together, I could use the Path class from System.IO which has a function that automatically stiches together a path with the appropriate connectors. I also discovered that C# had range slicing for substrings like Python did, and the code was far shorter to use, and as such I would be using the range operator going forward. To simplify the access of opening the actor or dialogue database to external classes, 2 public functions were made to fill in the required data into the OpenDb function.

A computer screen shot of text

Description automatically generated

Now that other scripts could easily setup a connection with the dialogue or actor database, it was time to start implementing the .dlg files. When the program had to go through the file, it first needed to preprocess it and find where the dialog sections were on the lines of the file. This also needed me to be able to read the text within the file.

Unity has a built in asset type called TextAsset, and the useful property of it was that the text could easily be extracted from within it with a single property. Writing code to sift through the lines and add it to a list of the line number and dialog section name was not terribly difficult.

A screen shot of a computer code

Description automatically generated

It consists of just a simple for loop, checking whether the end and start of a line had square brackets, the defining feature of dialog section headers. The code was extremely simple, but only if the .dlg file was to be recognised as a TextAsset. Unity unfortunately, was unable to do so, even with many attempts, including attempting to write a ScriptedImporter for the filetype.

A screen shot of a computer code

Description automatically generated

This was standard code to do the following, albeit without the icon (this was added as an easier check to see if the file was imported correctly), but unfortunately it would not work. Even after scouring for a solution to this problem, none were found, and as such I had to go for the less memory-efficient and slightly worse approach of using the C# method to opening and reading file contents – StreamReader. While the File class exists, it turns out to be far slower to StreamReader on its own, and as such I went with the below changed code to accommodate StreamReader.

A screen shot of a computer program

Description automatically generated

The new code is very similar to the previous, but has a custom counter instead, and reads off a new line using StreamReader.ReadLine(). It stops when the reader detects it has reached the end of the file.

### Saving and Loading System

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

## Prototyping

## Modules

## Comments

## Naming

## Validation

## Reviews

# Development Testing

## Evidence

## Fails

# Evaluative Testing

## Evidence

## Usability Tests

# Evaluation

## Success Criteria

## Usability Features

## Limitations

## Maintenance

# Bibliography

Anon., 2024. [Online]   
Available at: https://tf2sr.github.io/TF2SR-Wiki/

Atwood, J. & Spoisky, J., 2008. *Stack Overflow.* [Online]   
Available at: https://stackoverflow.com/  
[Accessed 26 June 2024].

d'Apprentissage, T. d. F. e., n.d. *Orthogonal Unit Differentiation.* [Online]   
Available at: https://tecfa.unige.ch/tecfa/maltt/VIP/Ressources/Articles/Game%20Design%20Patterns/collection/Alphabetical\_Patterns/OrthogonalUnitDifferentiation.htm#:~:text=When%20Units%20in%20a%20game,with%20different%20types%20of%20abilities.  
[Accessed 24 July 2024].

Electronic Arts Inc.; Respawn Entertainment, 2018. *Titanfall 2.* [Online]   
Available at: https://www.ea.com/en-gb/games/titanfall/titanfall-2

Frei, D., 2022. s.l.: s.n.

gskinner, n.d. *RegExr.* [Online]   
Available at: https://regexr.com/  
[Accessed 27 August 2024].

Half Life Fandom, 2007. *Overwatch Soldier.* [Online]   
Available at: https://half-life.fandom.com/wiki/Overwatch\_Soldier  
[Accessed 17 September 2024].

Huertas, R., n.d. s.l.: s.n.

id Software, 2016. *DOOM.* [Online]   
Available at: https://bethesda.net/en/game/doom-2016

id Software, 2020. *DOOM Eternal.* [Online]   
Available at: https://mods.bethesda.net/doometernal

Iyer, A., 2018. *Titanfall 2: How Design Informs Speed,* s.l.: s.n.

Leprawel, n.d. *Parkour Controller Script,* s.l.: s.n.

Microsoft, 2024. *C# Language Documentation.* [Online]   
Available at: https://learn.microsoft.com/en-us/dotnet/csharp/  
[Accessed 6 July 2024].

Official, T., 2016. *Titanfall 2: Inside Development,* s.l.: s.n.

OMOCAT LLC., 2020. *Omori.* [Online]   
Available at: https://www.omori-game.com/en

One More Level; Slipgate Ironworks, 2020. *Ghostrunner.* [Online]   
Available at: https://ghostrunnergame.com/

Patala, A. "., 2020. *Ultrakill.* [Online]   
Available at: https://store.steampowered.com/app/1229490/ULTRAKILL/

Valve Corporation, 1998. *Half Life.* [Online]   
Available at: https://www.half-life.com/en/halflife

Valve Corporation, 2024. *Steam Hardware Survey.* [Online]   
Available at: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam  
[Accessed 19 September 2024].

Valve Developer Community, 2005. *Category:AI - Valve Developer Community.* [Online]   
Available at: https://developer.valvesoftware.com/wiki/Category:AI  
[Accessed 18 September 2024].