UNIVERSITY OF BIRMINGHAM

School of Computer Science

COVID Destroyer: A 2D Game Built on Unity

Munir Suleman
1348560

MSc Computer Science
Supervised by: Dr Ahmad Ibrahim

September 2021

Acknowledgments

I would like to thank my project supervisor, Dr Ahmad Ibrahim, at the School of Computer Science at the University of Birmingham for the advice and guidance throughout the project. I would like to thank Priya Manimaran as the project inspector. I would also like to thank my parents and family for all their support not only throughout the project, but throughout this course. This accomplishment would not have been possible without them. Finally I would like to thank my friends for play testing the game during the development process and providing useful feedback.

Abstract

The on-going COVID-19 crisis has caused nationwide lockdowns, leading people to pick up new hobbies, one of which has been gaming. The gaming industry has already seen massive growth for the past decade, as consoles and desktops are much more readily available with hardware capable of experiencing new worlds through games. Along with this rise comes the resurgence of retro games. COVID Destroyer is a modern take on an old-school 2D arcade style shooting game, played from a top-down perspective. Throughout the project special care was taken in ensuring both software and game development principles were utilised alongside architectural design patterns. The game was developed using the Unity Engine and was built for Windows, macOS and Linux. The details of the classes, game objects and game mechanics are illustrated, as well as the use of Unity's built-in testing methods. COVID Destroyer was play-tested by users both for testing and evaluation purposes, revealing the games high usability and performance.

Keywords: Unity, C#, 2D-Game, UML, game design, testing

Table of Contents

List of Figures

1.0. Introduction

The aim of this project was to design and develop a desktop 2D game with themes revolving around the current COVID-19 crisis. Throughout the project a combination of software and game development processes were utilised. The Unified Modelling Language (UML) (UML, 2015) was used to demonstrate the design and structure of the software. The game was developed utilising the Unity Engine (Unity Technologies, 2005a), using C# as the scripting language. The software was targeted to be run on Windows, macOS and Linux.

The COVID-19 pandemic has caused on going issues globally concerning people's health, education, and the economy. Quarantine measures and country wide lockdowns have led to people being confined to their homes for weeks or months (López-Cabarcos *et al*., 2020). For most people, this has been an opportunity to be creative, learn a new skill or pick up a new hobby. One such hobby has been gaming. As of June 2020, a study showed an average increase of 39% on time spent playing video games during the pandemic globally (Simon-Kucher & Partners, 2020). However, this rise in gaming is not purely due to the lockdown measures. The gaming industry over the past decade has seen immense growth, with the global market generating close to $160 billion in 2020 alone (Wijman, 2020).

The early years of the gaming industry saw only two platforms (arcades and consoles), where the games were limited to what is now referred to as retro-games (Roth, 2020). Video games now more than ever are easily accessible through PC's, consoles, and mobile devices, and all platforms come with a wide variety of game genres (Wallach, 2020). Along with this comes the recent resurgence of retro-gaming with old consoles being re-released and classic games being ported to newer consoles (Heubl, 2020). An increase in the number of serious games, prolifically for educational purposes, has meant games are not purely for entertainment purposes (De Lope and Medina-Medina, 2016).

A video game is defined as a software application in which the player or players make decisions and carry out actions by controlling game objects or resources, in order to complete a goal (Salen and Zimmerman, 2003). It stands to reason the steps involved in game development would be identical to that of software development. However, due to game development's multidisciplinary nature, practices tend to differ from traditional software development (Aleem, Capretz and Ahmed, 2016). Alongside this, the fundamental difference

that games are products of entertainment, whereas traditional software aim to solve a problem or increase efficiency (Kasurinen, 2016). None the less, the basic principles of good software engineering practices are implemented within the gaming industry and throughout the development of this game.

This report aims to detail the process of the design and development of COVID Destroyer, including both software and game development principles and overall good engineering practices. This report comprises of the following chapters:

Background: An in-depth investigation into software and game development processes and the differences between them, game engines and different tools of implementation and the elements involved in game design.

Project Specification: A brief project overview followed by the requirements engineering (functional and non-functional requirements), the tools used for the project with emphasis on the Unity Engine, details on the story of the game and ending with a use case diagram.

Solution Design: The design objectives, followed by an insight into the system architecture, the scenes and views that make up the game, class diagrams detailing the underlying structure and the project management.

Solution Implementation: A detailed account of the main features and mechanics of the game and how they were implemented.

Testing and Evaluation: Details on the testing methods used throughout the project, user testing and the feedback given by users.

Discussion: Examining the implementation and design of the software and the potential for future work

Conclusion: Summary of the project.

2.0. Background

2.1. Software Development

IBM defines software development as a “set of computer science activities dedicated to the process of creating, designing, deploying and supporting software” (IBM, 2019). Other definitions differ slightly, most notably by adding testing after the design phase (ITChronicals, 2018). Software development is thus the systematic process of creating a software from idea through to deployment. The stages of this process are specified within the software development life cycle (SDLC). Several SDLC models exist and can be categorised as linear, iterative or a combination of the two. The most common models are agile, waterfall, and rapid application development (RAD).

2.1.1. Software Development Life Cycle (SDLC)

The SDLC consists of many stages (ISO/IEC 12207, 2008):

- Planning and Requirement Analysis: Performed by experts in various domains to gather information from the customer and market about what the product should be, how it will work and its purpose.
- Defining Requirements: Clearly define and document the product’s functional and non-functional requirements, typically within a software requirements specification document.
- Design: Design of the system architecture with reference to the requirements previously set. The architecture defines the functionality of each module, the flow of data and the communication between internal and/or external modules.
- Implementation/Build: Software design is translated into the source code and build of the product.
- Testing: Developed software is tested thoroughly for defects, are then fixed, and retested until the product reaches the standards defined in the documents.
- Deployment and maintenance: Released formally into the market. Once released, maintenance is performed to change elements of the product according to future needs.

The naming conventions and the number of stages differ depending on the source, but the underlying phases are the same.

2.1.2. Waterfall Model

In this model, the output of the previous stage is the input of the next stage making it a linear model. Stages only begin when the previous stage has completed (Sommerville, 2011). Due to its linear nature, this method is easy to understand and manage. Each step has easily identifiable deliverables, thus ensuring completeness before moving to the next stage. The waterfall model needs all requirements to be well defined and stable before advancing to the design stage, thus making it unsuitable for projects with uncertain requirements. It is also often slow due to its rigid structure and costly if changes need to be made to previous steps.

2.1.3. Agile Model

The agile model is a combination of the iterative and incremental model. The product is broken down into smaller sections, where each build adds further functionality. Testing is done throughout each build to minimise risks. The main advantage of agile is its flexibility. As an iterative model, new features can be added with ease and feedback can be taken after each increment. There is more open communication between teams and clients resulting in a faster release at a lower cost. However, unlike the waterfall model, there is a significant lack of documentation. User's also play a huge part in providing feedback and thus a large time commitment is needed from the user's perspective (Easterbrook *et al.*, 2008).

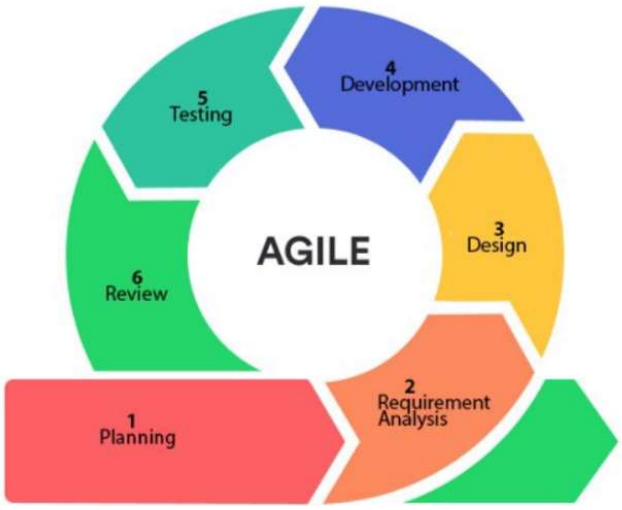


Figure 1: The agile model (Sandeepa, 2021)

2.1.4. Rapid Application Development (RAD)

The RAD model is based on prototyping and iterative development. It uses minimal up-front planning and favours prototyping for early testing ensuring each prototype is reusable. It reduces project risk by breaking it down into smaller segments, allowing all aspects to be

changed easily throughout the development process. Like agile, active user participation is necessary. Due to the lack of pre-planning, requirements may change frequently and significantly. Also with inadequate problem analysis, a situation may occur such that only the most obvious requirements are met, resulting in inefficient practices being built into the system (Ben-Zahia and Jaluta, 2014).

2.1.5. V-Shaped

Also known as the verification and validation model, the V-shaped model is a take on the waterfall model with rigorous amounts of testing at each stage. The testing ensures a well-documented working model at each stage of the development process. Like the waterfall method, it is easy to manage due to its rigidity. Along with the waterfall model's disadvantages, the V-shaped model is not ideal for ongoing projects or for complex object orientated projects (Ruparelia, 2010).

2.2. Game Development

Game development refers to the entire process the game goes through from initial concept to launch. Game developers implement the ideas created by game designers. There are three main stages to the game development process: preproduction, production and postproduction (Kanode and Haddad, 2009).

Preproduction begins when the idea for the game is envisioned. The concept of the game is fleshed out at this stage before any design or programming takes place. Basic questions are answered: What type and genre of video game is being produced? Is it 2D or 3D? What platform will it be built on? Will the game engine need to be created or can an existing one be used? What are some of the main characteristics, game mechanics and story elements? What is the goal the user is trying to achieve? All these questions and more are answered within the concept document, where the goal and purpose of the game are detailed. It puts ideas in context with what is possible. The purpose of this is to sell the game to potential producers who would fund the project. This document would also include information about the target market and existing competition (Novak, 2007).

One of the most important documents throughout game development is the game design document (GDD). Before creating the GDD, requirements need to be gathered. For original games users do not yet exist, so cannot be questioned about potential features. Instead the

entire development team create the requirements from the initial ideas, which they then communicate to others through the use case diagram in the Unified Modelling Language (UML) (Bethke, 2003). The GDD is the blueprint from which the game is built. It specifies the rules of playing the game, the interface, the world, or level plan, as well as the game play, characters and their abilities or items, and danger elements such as hazards or enemies. The GDD also includes technical elements such as hardware requirements, programming language, and the limitations of the game engine. The complete document is used as a reference guide for the rest of the game development process (Rogers, 2014).

Prototypes are also created in the preproduction phase. Story boards, concept art and interface mock-ups guide the visual style the game aims for. The environment, characters, control schemes and other in-game elements are also modelled. At this stage all teams (programmers, designers, artists etc.) collaborate to decide what is and what is not viable. The main outcome of the preproduction phase is the GDD and the project plan which outlines the path that should be taken whilst developing the game. This includes the resource plan, budget, schedule, and milestones in order to keep the project on track (Novak, 2007).

The production stage is the predominant part of the development process and takes majority of the time, effort, and resources. The UML class diagrams are created, describing the static design of the software, how classes are associated with one another and how they are organised (Bethke, 2003). Video games are an amalgamation of different technical and artistic skills combined into one project. The artists work on the graphics, 2D/3D art for the characters, world assets and the environment, visual effects, and the user interface. Level designers are responsible for structuring the world in which the user will be interacting, planning obstacles and objectives for each level. This information is found in the GDD. The programmers create the playable product backbone on the game engine. Sound designers create the audio for the game which is often dynamic as it depends on the state of the game and the user's actions (Novak, 2007). This stage is often an iterative process, filled with trial and error and adjustments to the GDD. The game is tested internally to ensure characters act as designed, and the system is stable. Bugs are detected and resolved. The result of the production phase is the complete game (Novak, 2007).

Postproduction focuses on user testing and the various version releases of the game. Games are first released in their alpha and beta stages to detect any issues with the system and to

obtain direct feedback via user testing. Adjustments are made before the launch of the product. The lifecycle does not necessarily end there, with various productions periodically adding downloadable content (DLC) or constantly updating the game to extend its lifespan, keeping it new for their player base (Novak, 2007).

These three stages can be deconstructed into concept, plan, design, develop, and test, where concept and plan fit within preproduction, design and develop fit within production, and testing is considered as postproduction (O'Hagan, Coleman and O'Connor, 2014). They can also be split into concept, preproduction, prototype, production, alpha, beta, gold, and postproduction. Alpha and beta are near complete builds requiring some artwork or bug fixes, and gold is the final product sent for manufacture (Novak, 2007). In either case, the main stages highlighted above are incorporated within the development process. Other game design life cycles are detailed by (Ramadan and Widyani, 2013).

The development of a game often involves a number of people from different disciplines. Although on the rare occasion games have been made by an individual, for example the original Prince of Persia (Mechner, 1989) or more recently Stardew Valley (Barone, 2016), the typical development team consists of programmers, artists, game designers, sound designers, and testers (Rogers, 2014). Programmers write the code that displays the graphical elements to the screen, develop the control system that allows the user to interact with the environment, and writes the AI systems that control the enemies and world objects among many other tasks. Programmers often use placeholder art when developing the game. The artists create the visuals for the game that would replace the placeholder art as well as the animations. The game designers produce the ideas and the rules that make up the game. The goal of the designer is to make the game "fun" for the users and should design based on this principle. The sound designer collaborated with the composer to create the music and sound effects for the game. This role is especially important as sounds often deliver a great deal of information to the user and is thus an important part of the development. Finally the testers often are responsible for quality assurance via rigorous play testing throughout every aspect of the game in order to identify any bugs or defects (Rogers, 2014).

Game development and game design are often mistaken as being the same. Game design refers to the conceptual side of the software involving the initial vision, the game mechanics, the story, the characters, the world, etc. Developers implement these ideas into the program.

In essence the designers create the initial framework. Developers turn these visions into reality by programming the concepts into a functional video game (Bethke, 2003).

2.3. Differences in Software and Game Development Processes

One of the fundamental differences between traditional software and game development is the aim of the application being created. Traditional software target productivity or efficiency, whereas games aim to provide an experience for the user (O'Hagan, Coleman and O'Connor, 2014). As a result, usability testing is not always applicable for games as increased difficulty levels are one of the features that challenge the user, which is essential to a game. User experience (UX) design for traditional software development is centred around the user and delivering a pleasant experience for anyone interacting with the product, thus streamlining any difficulties the user may have in navigating the system. Game design on the other hand, is based on the philosophy of putting the player at the centre of the development process. The user for a video game is willing to put in more effort and time in tackling the challenges that are presented to them. These challenges are partly responsible for constructing a "fun" and engrossing game.

The main differences between the two development processes stem from the preproduction phase. Traditional software development teams put more emphasis on requirements engineering compared to game development, where instead the GDD is considered the blueprint from which the game is designed. Requirements gathering in traditional software development, takes the customer's needs and builds the functionality of the software from this. However, games are not built primarily around user requirements, but around entertainment value. Regardless, game development can benefit greatly from requirements gathering by reducing the number of iterations needed and preventing unnecessary late addition of features or game assets (Kanode and Haddad, 2009).

Due to the extensive amount of multimedia assets (artwork, sounds, visual effects, etc.), prototyping for games involve storyboards, concept artwork, user interfaces, characters and more (Kanode and Haddad, 2009). Traditional software prototyping is most useful for systems with high level user interaction, for example mobile apps. It thus stands to reason, prototyping for game development is a much larger component in preproduction in comparison to software development.

UML diagrams for software systems communicate the design and structure visually. In game development use case and class diagrams are common, as in most cases the other diagrams are not as relevant (Bethke, 2003). Despite these differences the game development process can use proven software engineering principles to solve problems common in game project management (Aleem, Capretz and Ahmed, 2016).

2.4. Brief Evolution of Gaming and Game Genres

Early arcade games like Asteroids were rendered in vector graphics before the development of raster graphics. With the use of coloured pixels came cartoon-inspired character games, such as Pac-Man (Namco, 1980) and Donkey Kong (Nintendo, 1981). During the mid-1980s, arcade games saw a significant rise along with game genres and themes having a greater variety. Cabinets and cockpits filled arcade rooms around the world with classic game titles still played to this day. Home systems in the 1990s rivalled and surpassed arcade machines and controlled the market with consoles like the PlayStation and Xbox. As processing power increased and the size required to hold such power decreased, hand-held portable gaming devices, like the Gameboy and PSP, became popular (Rogers, 2014) and eventually evolved into games on mobiles, which currently dominate the gaming industry revenue. However, recent years have seen classic games return in different forms via re-released consoles or ported onto desktops (Heubl, 2020).

The genre of the game describes the style of gameplay. Games within the same genre tend to share similar characteristics. There are many genres and subgenres for video games and this list is continuously growing as designers merge and invent new styles. The unique genre's require different skills or levels of interactivity from the user (Rogers, 2014). Action games typically require hand-eye coordination, for example shooters, fighting games, and platformers. Adventure games, such as role-playing games (RPGs), require elements of puzzle solving and world exploration to collect items. Strategy games involve planning and are often turn based. All games contain elements that would class them as one type of game or another, making it hard to purely categorise video games (Pavlovic, 2020). For example Space Invaders (Taito Corporation, 1978) is typically categorised as an action shoot-'em-up, but also requires strategy from the user to succeed.

Since the COVID-19 pandemic, the increase in the amount of video games being played has been significant. Along with this comes the numerous game titles with COVID or virus-related themes within both the video and board game industries. These games have provided useful information to the users regarding public health, as well as providing some cathartic comfort for people against a struggle they face each day in the current situation (Hay, 2020). Fauci's Revenge (Beat The Bomb, 2020) and COVID Invaders (Ternyak and Peysakhovish, 2021) are examples of games created using the pandemic as their main theme; both taking inspiration from Space Invaders.

2.5. Game Engines

A game engine is a framework that provides developers with a range of different tools, utilities and interfaces that can be manipulated to build the game. The engine abstracts these tools by hiding the low-level details of various tasks that make up the program (Sherrod, 2007). This abstraction allows the same game to be played on various platforms. Often the game engine incorporates specialised game middleware for physics, sounds or videos. Physics components are the most significant when it comes to game engines, where the game objects must follow the rules set by the physics simulation.

A game differs from a game engine. The game is made up of the content, the characters, the behaviours of real-world objects and more. Whereas the game engine components include rendering, animation, collision detection, physics, inputs, and the graphical user interface (GUI) (Paul, Goon and Bhattacharya, 2012).

2.5.1. Unreal Engine

One of the most popular game engines, the Unreal Engine (Epic Games, 1998) allows for cross-platform creation and supports programming in C++. Over 25% of games on Steam are built on the Unreal Engine (Toftedahl, 2019). As an open-source software, the game engine's code can be modified, allowing developers to change the engine to suit their purpose.

The Unreal Engine is regarded highly for its graphics quality. The gaming world is headed towards photorealistic graphics, especially in AAA titles, making realistic visuals a highly sort after quality within a game engine. C++ is not the easiest programming language to learn however, decreasing its ease-of-use for beginners. As a result it is more suitable for game development teams rather than first-time game developers or individuals (Zarrad, 2018).

2.5.2. Unity

Unity (Unity Technologies, 2005a) is the most used game engine in the world particularly for indie game developers. Close to 50% of games releases on itch.io are built on the Unity Engine (Toftedahl, 2019). The most recognised benefit is its ease of use. It allows for cross-platform creation for effortless porting and is programmed using C#. Unity allows porting to 30 different platforms, making it simpler to build games for multiple platforms through just one build (Dillet, 2018). The engine has wide support through its community and documentation.

Another one of the engine's highlights is 2D game development. Its drag-and-drop interface, asset store and intuitive design makes game development easier, especially for new developers. However, Unity is a proprietary software which prevents developers from studying or modifying the engine's source code. (Dar, 2021).

2.5.3. Godot

Godot (Linietsky and Manzur, 2014) is an open-source engine capable of creating both 2D and 3D games. The engine provides a large range of tools, a simple user interface and its own scripting language (GDScript), making it easy to use and beginner friendly. The engine developers provide constant updates, and the community can provide a great deal of support. However, Godot's biggest drawback is its weak 2D physics engine (Wirtz, 2021).

2.5.4. CryEngine

CryEngine (Crytek, 2002) is a multi-platform engine, most successful at photorealistic visuals. It provides developers access to the source code, allowing developers like Ubisoft to modify the engine to their needs and keep an in-house copy of the modified version. CryEngine is easy to learn with simple user interfaces and allows for fast iteration process due to having no build or compile time in its editor. However, the documentation and smaller community, along with difficult out of the box tools, make it difficult to learn the engine for new game developers (Dealessandri, 2020). The engine is especially known for developing first person shooter games and its VR support.

2.6. Architecture and Programming Patterns

The architecture of a system can be tied directly to its requirements or desired attributes. These quality attributes are as follows: availability, modifiability, performance, security, testability, and usability (Bass, Clements and Kazman, 2003). Depending on the product the

importance of these qualities differs. For example, for game architecture, performance and usability are far more crucial than security. Therefore, the main goal of the system architecture would be dependent on the software.

Programming patterns are reusable solutions to common problems. They are a template of how to solve the problem in many different situations and are thus elegant and optimal solutions (Gamma *et al.*, 1994). Implementing such patterns often add flexibility to the system for future development. Along with this flexibility comes decoupling, which means changes to one part of the software does not necessitate a change to others. Of course complete decoupling is impossible, but reduced coupling leads to good system architecture. Good architecture requires effort and comes at a cost (Nystrom, 2014).

Software architecture with modifiability in mind results in making the program flexible, minimizing the work and effort required to update the application. But for games this hinders the performance, as patterns that improve flexibility often come with runtime cost. Therefore there must be a trade off when building the software regarding good architecture for easier implementation over the lifetime of the project, wanting faster runtime performance, and adding decoupled features to the system with ease (Nystrom, 2014). Implementing programming patterns often assists the ease in which the new features can be added.

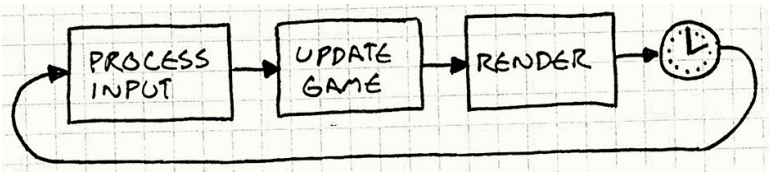


Figure 2: The general game loop (Nystrom, 2014). *Game loop differ depending on the engine used.*

A quintessential game programming pattern is the game loop (Figure 2). The game loop runs continuously during gameplay, where it processes user input, updates the game state, and renders the game. Most game engines run the game loop without the need of the programmer to directly code the pattern (Nystrom, 2014). The game loop is responsible for keeping a consistent frame rate every second (FPS). A delay is added if the game loop runs faster than the desired frame duration to keep the FPS constant. However, the game slows down if the loop is too slow to complete all the processes required within the frame, thus hindering performance (Dickinson, 2017). Implementing other patterns may also affect the performance of the system in different ways.

Patterns fall into different categories: Design, sequencing, behavioural, decoupling and optimisation. The game loop falls under the sequential pattern and is essential for every game (Nystrom, 2014). Other patterns are used when needed. Some examples of programming patterns are highlighted below.

The singleton pattern ensures the class only has one instance and provides a global access point to it. The class is initialised at runtime and the instance is not created until the game accesses it, saving both memory and CPU cycles. However, since it is a global variable, the singleton encourages coupling (Nystrom, 2014). The component pattern allows a single entity to span multiple domains by reducing it to a container of components. These components allow for a highly decoupled structure that can communicate with each other if required (Gamma *et al.*, 1994). The observer pattern allows for an observer to register for an event. The subject changes state and notifies any observers of this event. Through this loose coupling forms a one-to-many dependency between the subject and observers (Gamma *et al.*, 1994; Nystrom, 2014). Patterns assist both game and software developers structure the software for flexibility and modularity.

Patterns and principles are sometimes confused. As part of an overall strategy of agile and adaptive programming, object-oriented design principles were proposed. Programmers should apply these principles while working on software to remove potentially buggy code by refactoring until it is legible and extendible (Martin, 2000). The single responsibility principle requires a class should only have a single responsibility. The open-closed principle requires each entity should be open for modification but closed for modification. These two principles along with the Liskov substitution, interface segregation and dependency inversion make up the SOLID principles (Martin, 2000), which are considered the basis of creating clean and modular design in order to test, debug and maintain code. The implementation of these principles alongside the programming patterns ensures robust software architecture.

3.0. Project Specification

3.1. Project Overview

COVID Destroyer is a 2D, top-down view, action-shooter game, which more specifically falls under the shoot-'em-up genre. The story for the game revolves around the current pandemic, where the user attempts to vanquish the corona virus in each continent. The project used a combination of software and game development processes. Throughout the following sections of this report, the user refers to the person playing the game and the player refers to the on-screen character controlled by the user. The game consists of 6 levels, each set in a different environment. The goal for the user is to defeat the enemy on each level, avoiding its projectiles by moving the player side to side and shoot bullets of their own at the enemy.

The main mechanics of the game revolve around the enemy advancements on each level and the user's choices which change the player's weapon, health, shield, or movement speed. The difficulty increases each level by having the enemy employ elements of randomness, different activity modes and more advanced features enabling player targeting and bullet dodging.

The game was built as a desktop application for Windows, macOS and Linux. The game uses the Unity Engine and is programmed in C#. The system is mostly contained within the application, the only exception being the text files written in JSON format containing player information and high scores. The game has a PEGI rating[1] of 7 (PEGI, 2003), with a target audience age range of 7 – 40.

3.2. Requirements

Sommerville (2011) defines the functional requirements as detailing the services the system should provide and how the system should react to given inputs. The non-functional requirements describe the constraints which affect how the system would accomplish the functionalities. The requirements in game development are often included within the game design document (GDD). Other components that would be included within the GDD are included throughout sections 3, 4 and 5. The functional and non-functional requirements for the game are shown in Appendix A. Wireframes were made from these requirements (Appendix C).

[1] PEGI rating is the European video game system that confirms the content is suitable for certain age groups.

3.3. Unity Engine

Game engines provide the bases from which a video game is built. The game could have been built using various other engines, some of which were highlighted in Section 2.5. COVID Destroyer was built on the Unity engine using C# as the main programming language. Initially, Unity focused on delivering an engine better suited for independent developers (Dillet, 2018), by creating a software that is easy to use and has less of a learning curve in comparison to other game engines. This was accomplished through the graphical user interface (GUI), which allows developers to create GameObjects[2] (definition provided in Section 3.3.2), add components to GameObjects, change public script variables from the inspector, and much more. Unity also has built in libraries for physics (Nvidia's PhysX physics engine), rendering (OpenGL and DirectX) and audio (OpenAL) (Goldstone, 2009). Although Unity does not offer a means of manipulating the engine itself, it provides the tools needed to develop a wide variety of games.

Unity provides functionalities via the Unity Editor (GUI) to drag and drop assets and generate pre-built GameObjects with components attached. But for the purpose of this project, the use of these functionalities was minimised to develop skills in game programming and to better understand the architecture on which this game is built. Before developing the game, it is crucial to understand the terminology and environment (Goldstone, 2009).

3.3.1. Game Loop / Order of Execution / Life Cycle

Monobehaviour is the base class from which every Unity Script inherits. It provides user call-back functions that require execution at specific times within the lifecycle (Unity Technologies, 2005b). Monobehaviour inherits from Behaviour, which in turn inherits from Component. Any class that inherits from the Behaviour class can be enabled or disabled. The enabling/disabling of any behaviours is accomplished through the editor or via a script. The Component class is the base class of any script that can be attached to a GameObject. As a result, any script attached to a GameObject derives from MonoBehaviour (Goldstone, 2009).

As stated earlier (Section 2.6.), the game engine often runs the game loop autonomously, but provides users with access to a large collection of event messages via the MonoBehaviour class. This game loop, often referred to as the order of execution (Figure 3), allows code to be

[2] Unity refers to an object within a game as a GameObject (without the space).

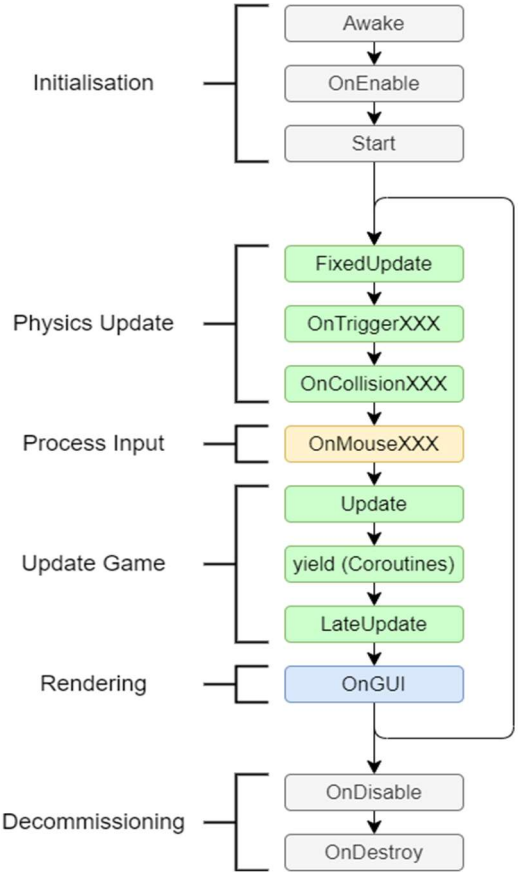


Figure 3: Unity's order of execution for event functions. Simplified from Unity manual

executed based on current activities in the designed system. Figure 3 only highlights the main events used for this project (Unity Technologies, 2005b). The game loop differs slightly from the conventional pattern shown earlier (Figure 2). The Unity engine updates the physics portion of the game states before processing the user input, updating the rest of the game states, and finally rendering the screen. There are also initialisation and decommissioning steps before and after the loop respectively.

The initialisation functions provide a means of preparing the component before entering the game loop. The Awake()[3] function is called once, when the component is created, regardless of the component being activated or not. The OnEnable() function is called when the object[4] is enabled. While Awake() would only run once when created, the OnEnable() is called each time the object is reactivated. Start() is called once the component is activated on the scene for the first time, before the game loop's first frame. Awake() always runs before OnEnable() and Start() and is therefore used to cache components or process items that are needed within the Start function, prior to entering the game loop (Dickinson, 2017).

A game loop runs once per frame. Update() is called every frame that the object is active, but time between calls may differ depending on the number of objects on the scene, the CPU and GPU loads, and the amount of processing the function is doing within the frame. FixedUpdate() is similar to Update, but is not dependent on the frame rate, resulting in its execution timing being more consistent. Once the object is deactivated, the decommissioning steps begin. OnDisable() is the opposite of OnEnable(), executing every time the object is deactivated. As the name implies, OnDestroy() is called when the object is destroyed explicitly

[3] For the purpose of this report, when referring to a function/method, brackets are used [e.g. Awake()]
[4] GameObject and object are being used interchangeably but refer to the same thing.

by calling Destroy(), or when the scene is unloaded. The cycle repeats itself when a new object is created. Understanding this life cycle is critical for increasing performance and ensuring stability of the game (Dickinson, 2017).

Due to the nature of the game loop, all code called inside Update() takes place all at once in one frame. However, this is not always ideal as some game logic may want to be executed over a number of frames. Coroutines (Unity Technologies, 2005b) allow a function to be paused and waits until a certain condition or action occurs before continuing (e.g. end of frame, wait a few seconds or wait for the completion of another coroutine). The Monobehaviour class is what allows a Coroutine to be started, stopped, or managed.

3.3.2. Terminology

Assets are the building blocks from which a Unity game is developed. Assets come in different forms: scenes, scripts, images, sound files, prefabs, etc. Essentially any object or file used to create the game is classed as an asset. A scene contains the world in which the game takes place. Each scene can be considered as individual levels or areas. Any asset placed within a scene must be wrapped in a GameObject. Scenes often contain many objects. GameObjects are at first empty containers, until components are attached to it. Components are isolated functionalities that give that functionality to the object it is attached to. They can create behaviour, define, or change appearance, influence the objects characteristics and much more. For example the Transform component when contained within the GameObject, allows movement along the three axes, rotation, and scaling of that GameObject. This report will draw attention to the components attached to various GameObjects.

Most components come pre-packaged within Unity with variables that can be changed within the inspector window of the editor. But often developers require custom components to configure the object to their own precise requirements. Scripts are components, which can be attached to objects, and can affect the appearance or behaviour of it like any other pre-built Unity component. GameObjects house numerous components and often have child objects. GameObjects can be stored, complete with components and configurations, to spawn them when required and are known as prefabs. Prefabs allow objects to have multiple instances with different property values, allowing developers to use a GameObject as a template and instantiate it multiple times with custom settings.

3.4. Other Tools

The Unity Engine (Unity Technologies, 2005a) is the main software used to develop the game. Other tools were used to enhance the development process and to produce a polished game. Visual Studio Code (Microsoft, 2015) and Visual Studio 2019 were the chose IDEs to write the C# code. PixelArt (Ware, 2021) is a free online pixel drawing tool and was used to create most of the game art. Other game arts are from artists found online and are free to use or licensable. These images were edited using Gimp image editor (Gimp Development Team, 1998). Audio for the game was found online from free to use sources and was edited using Audacity (Audacity, 2000).

3.5. Game Story

Rogers (2014) notes there are three type of people when it comes to stories in games: Users that want to get deep into the story, users that are into the story as it happens and those that pay no attention to the story. Developers have different views on the importance of stories in games. Casual games[5], like those played on mobiles, allow users to jump directly into the gameplay with no story driving the purpose of the game. But the purpose of a casual game revolves around short game sessions. The presence of a story would just extend game session without adding to the proposed user experience. Retro games, such as Pac-Man (Namco, 1980) or Tetris (Pajitnov and Pokhilko, 1984), often did not have a story beyond the general narrative created by the player. There are also many examples of story driven games (e.g. Life is strange (Square Enix, 2015), BioShock (2K Games, 2007)). The importance of a developed story intertwined with the gameplay of a video game is dependent on the design choices and the type of user experience the developers want to establish. In either case, the importance of not confusing story for gameplay and vice versa needs to be emphasised (Rogers, 2014).

The title of the game is a significant factor in marketing the game and grabbing the user's attention. COVID Destroyer is an example of a literal title (Rogers, 2014), where the name makes it easy to deduce the games properties or story. The story of the game revolves around the player travelling around the world attempting to destroy the corona virus. The current COVID-19 pandemic brings about a global enemy we are trying to defeat each day. With the appeal of echoing the current climate of the pandemic, COVID Destroyer provides users a

[5] Casual game refers to video games that do not need a major time investment to play.

cathartic, enjoyable activity, giving the user a sense of control in an uncontrollable situation. Therefore, the virus was chosen as the main enemy for the game designed in this project.

The “hero” is given the mission to defeat COVID around the world. On each level the player is presented with a new arena, each set-in a different continent. The six levels represent all the continents (excluding Antarctica). Each level starts by debriefing the user about the current situation of the virus in the current location and the enemy type they are about to face. The user must complete the game before the virus takes over the world.

3.6. Use Case Diagram

The UML use case diagram (Figure 4) summarises the interactions of the user and the game. Use cases specify the expected behaviour of the system from the user’s perspective. The diagram shows an outline of the functional interactions the user has with the game.

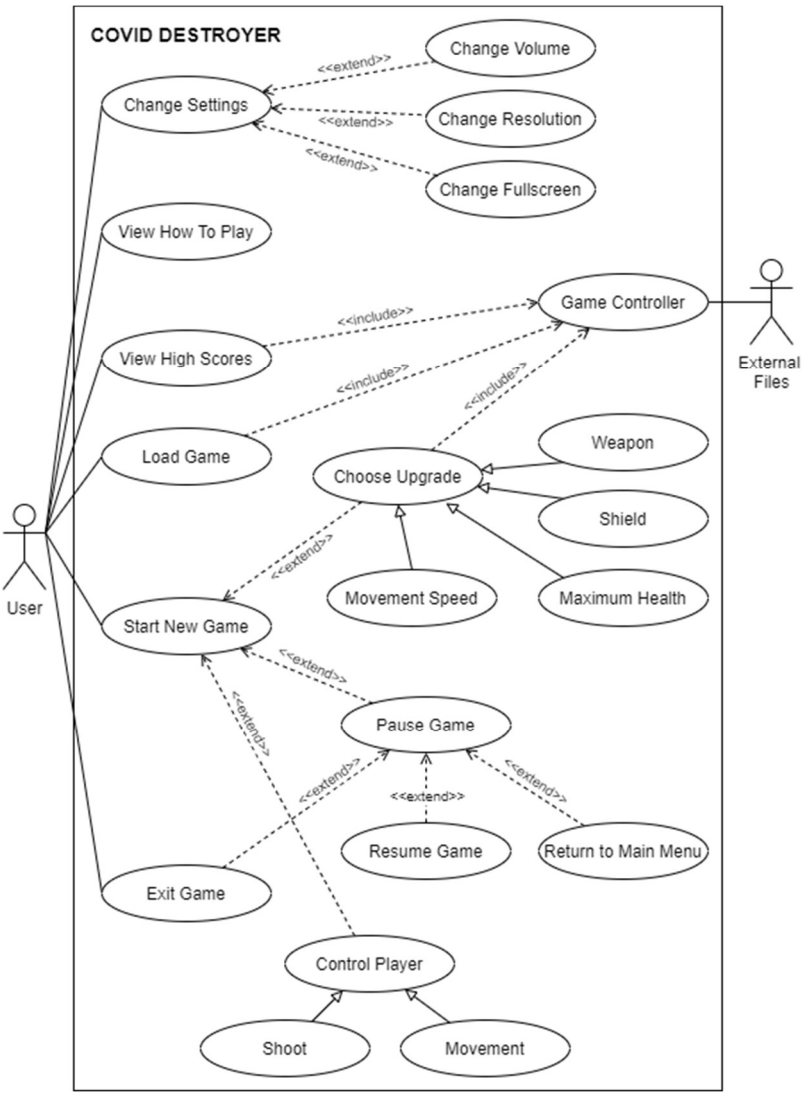


Figure 4: Use Case diagram for COVID Destroyer

4.0. Solution Design

4.1. Design Objectives

As stated in Section 2.6, the order of importance given to the attributes govern the architecture of the system. From the above requirements (see Section 3.2), the usability, performance, modifiability, and testability of the game, are the key attributes. The usability of a system is concerned with the ease at which the user is able to accomplish a task (Lundberg *et al.*, 1999). The usability in reference to a game is concerned with the user interface and the user controls rather than the actual gameplay, where difficulty in completing a level is seen as a challenge rather than a hinderance. As a result it was imperative that the system interface is visually consistent through its colour schemes and fonts, easy and intuitive to use, providing information to the user when needed to navigate through the game.

Performance concerns the timing of events in response to user interactions (Lundberg *et al.*, 1999). The system must perform without any crashes and ideally without any frames dropped, but this is largely dependent on the user's hardware. Throughout the development process, the modularity of the system's features is of utmost importance, whilst also considering the potential to scale the game and ensuring each feature can be modified with relative ease. Testing procedures must be put into operation to ensure system robustness. As a result, the game must be developed with testability in mind. Given the nature of the system, lesser importance is attributed to security and availability as these are more relevant for traditional software systems. In order to build a system encapsulating these main attributes and to meet the requirements previously set out, an appropriate architecture must be implemented alongside logical design structure.

4.2. Architecture

The Unity framework is designed around the component pattern. The component pattern states that if an entity spans multiple domains, these domains can be isolated by separating them into individual components and the entity acts an empty container holding these components (Nystrom, 2014). Let us take, as an example, a player. The player needs a visual aspect to represent itself on the screen. In 3D this would involve a mesh and material, whereas in 2D the player can be seen as an image. The player also requires an underlying

physical component demonstrated as a rigid body. For movement, the rigid body would need to be rotated or translated along a plane/axis. Furthermore, the player needs a be able to detect collisions, interact with the environment and perform attacks. All these domains are separated into individual components and attached to the entity.

This separation of domains into components present a different problem; how do components communicate with each other? The Entity Component System (ECS) (Nicoll and Keogh, 2019) is a software architecture pattern widely used for game development. The pattern consists of three parts: entity, component and system. The entity is an object with no actual data or behaviour. In Unity an empty GameObject is considered as the entity. The component holds the data associated with the entity. The component itself has no behaviour but can provide behaviour and appearance, to the GameObject. The system provides the logic and functionality that transforms the component from its current state to its next state (Unity Technologies, 2005b). The system combines the data received from multiple components to carry out required tasks. In essence the components do not need to communicate with one another because the system carries out all the logic and functionality, further reducing the need for coupling. ECS follows the principle of composition over inheritance.

Inheritance and composition dictate the relationship between classes and objects. Inheritance derives a class from another, resulting in classes being tightly coupled as changing the parent class has an impact on all of the children classes. On the other hand, composition defines a class as the sum of its parts, producing loosely coupled code improving flexibility (Gamma *et al.*, 1994). With this flexibility comes modifiability, which is why composition was chosen for developing COVID Destroyer.

Cook (2020) states that Unity uses ECS architecture as the basis for their engine. In actuality, Unity uses the Entity-Component (EC) architecture as built-in components in Unity provide some functionality along with the data and hence are a combination of a component and system. Scripts can reference entities and their components via Unity's functions to retrieve information.

The developed game employs the EC architecture. This helped to defragment and better organise the codes architecture. By utilizing a component-based pattern for the game, scripts were developed with modularity and performance in mind by providing smaller, more specific

functionalities within a class. In addition, individual components could be analysed, improving the testability of the system.

4.3. Scenes and Scene Hierarchy

The game consists of 8 scenes: Story scene, title scene and six level scenes. All of these scenes contain the same base level hierarchy GameObjects (Figure 5). Unity uses the concept of parent-child hierarchies to group GameObject within a scene. Child GameObjects inherit the movement and rotation of the parent object through the transform component. The hierarchy window within the editor helps structure the entities within the game.

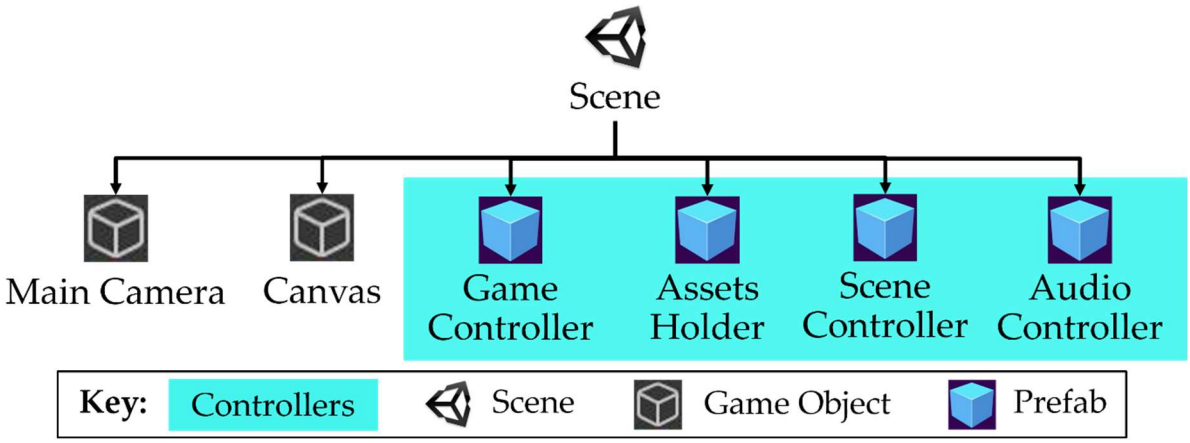


Figure 5: The base hierarchy of all the scenes in the game

The main camera shows the user's perspective of the world as seen on the screen. For example, in a game with a scrolling screen where the game follows the player, the user would only see a portion of the world at any given time. The camera would be attached to the player and depending on the size of the camera only show the world within a given radius. In COVID Destroyer, all views (see Section 4.4) are confined to the limits of the main camera, which is fixed. The camera is added to the scene automatically by Unity when the scene is created.

The scene is also made up of a canvas and four controllers (see Section 5.7 on controllers). The main canvas is the parent object to the background and all the views[6] in each scene. A script was attached to the canvas that creates the GameObjects as its children and controls which of these views is active. The canvas script differs in each scene, populating the game with different objects depending on the state.

[6] As with all entities in Unity, the views are all GameObjects.

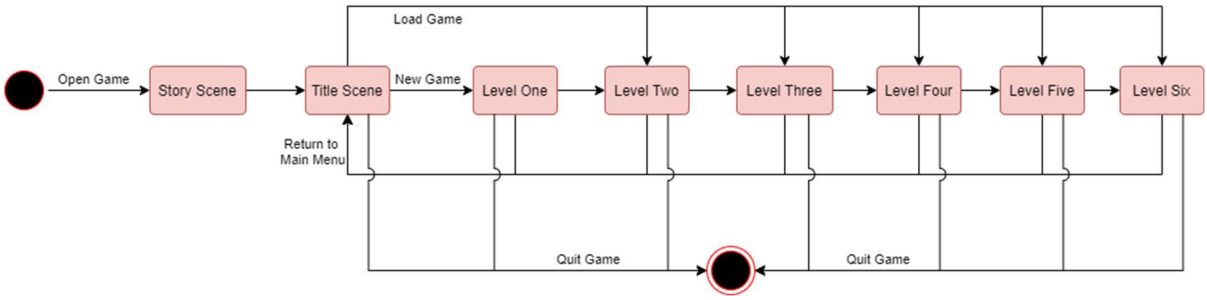


Figure 6: State diagram of all the scenes

The game has a relatively linear flow (Figure 6). Upon opening the application the user is first met by the story scene, where they then transition to the title screen after giving the prompted input. The title scene can transition to all the levels depending on the user's choice to either start a new game (transitions to Level 1 scene) or load a game (transition to the other level scenes). The user can return to the title scene from any of the levels and can quit the game from all scenes barring the story scene.

4.4. Views

For the purpose of this project, views are defined as the different screens the user is presented with. The views are all children of the main canvas. The number of views as well as their content differ from scene to scene. The story scene only has one view: Story View. The title scene consists of: Main, Settings, Quit, How to Play, High Scores, New Game Confirm, Load Error, and Enter Name views (Figure 7A). Although the content of the levels differs, the views are the same in each level and consist of: Starting, Game, End, Pause, Game Over, Return to Main and Quit (Figure 7B). Sections 5.8 and 5.9. detail the content of these views.

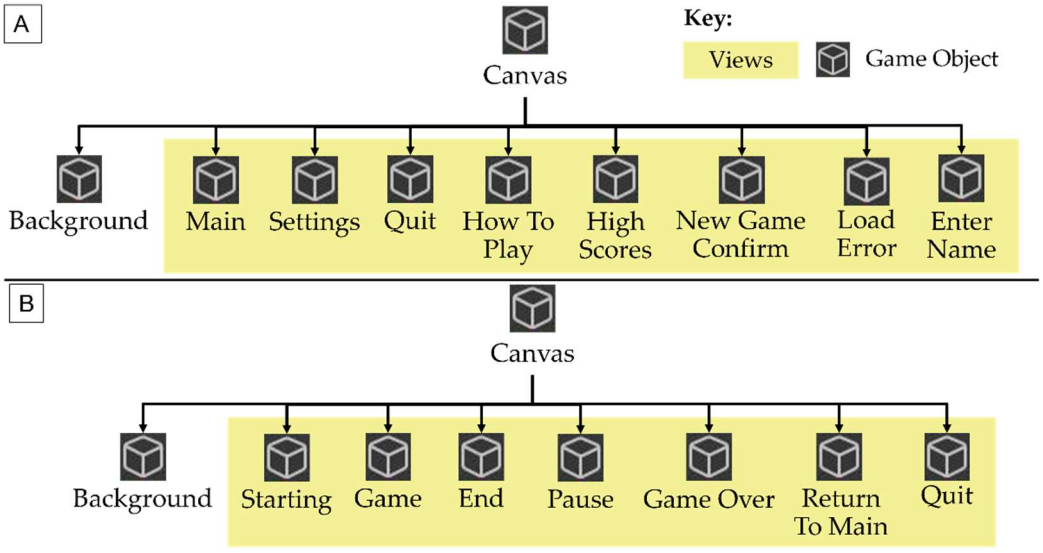


Figure 7: Canvas hierarchy showing the views of the title screen (A) and the levels (B)

4.5. Class Diagrams

The UML class diagram is a static design diagram that describes the structure of the system. It shows the classes, their attributes and methods, and the relationships between the objects. It is considered to be the most effective static diagram at communicating the classes and the relations between them, as others are merely variations of the class diagram (Bethke, 2003). However using the entity-component pattern and composition makes it difficult to draw a traditional class diagram.

An example of a traditional class diagram is drawn by Edwards, Li and Wang (2015) for their system built on Unity. Besides showing the inheritance of Monobehaviour for each of the classes, it does not communicate the relationship between the objects, which is one of the purposes of a class diagram. Therefore, there are some differences in illustrating the class diagrams for the game (Figures 8, 9 and 10). Since scripts in Unity are considered to be components as they can be attached to GameObjects, components were treated as classes and the GameObjects as entities which are composed of a number of components. The GameObjects are coloured blue, built-in Unity components are in orange, and classes created specifically for COVID Destroyer are in white. For the Unity components, only those properties that were used in the making of this project are labelled. The diagrams clearly show the structure of the levels for the game through entities and components, and how composition was used to build the game.

The level scenes class diagrams below are separated into multiple parts for readability purposes. Figure 8 shows the classes and components at the base level of the scene hierarchy, detailing the properties and methods of the controllers. Figure 9 features the classes that make up the main canvas and the views excluding the Game View as this is shown in Figure 10. Since a GameObject often contains only one instance of a component, most multiplicities are 1 to 1 and are not labelled on the diagram to reduce clutter. Multiplicities are labelled on the diagram when they are not 1 to 1 (e.g. multiple audio sources for the audio controller).

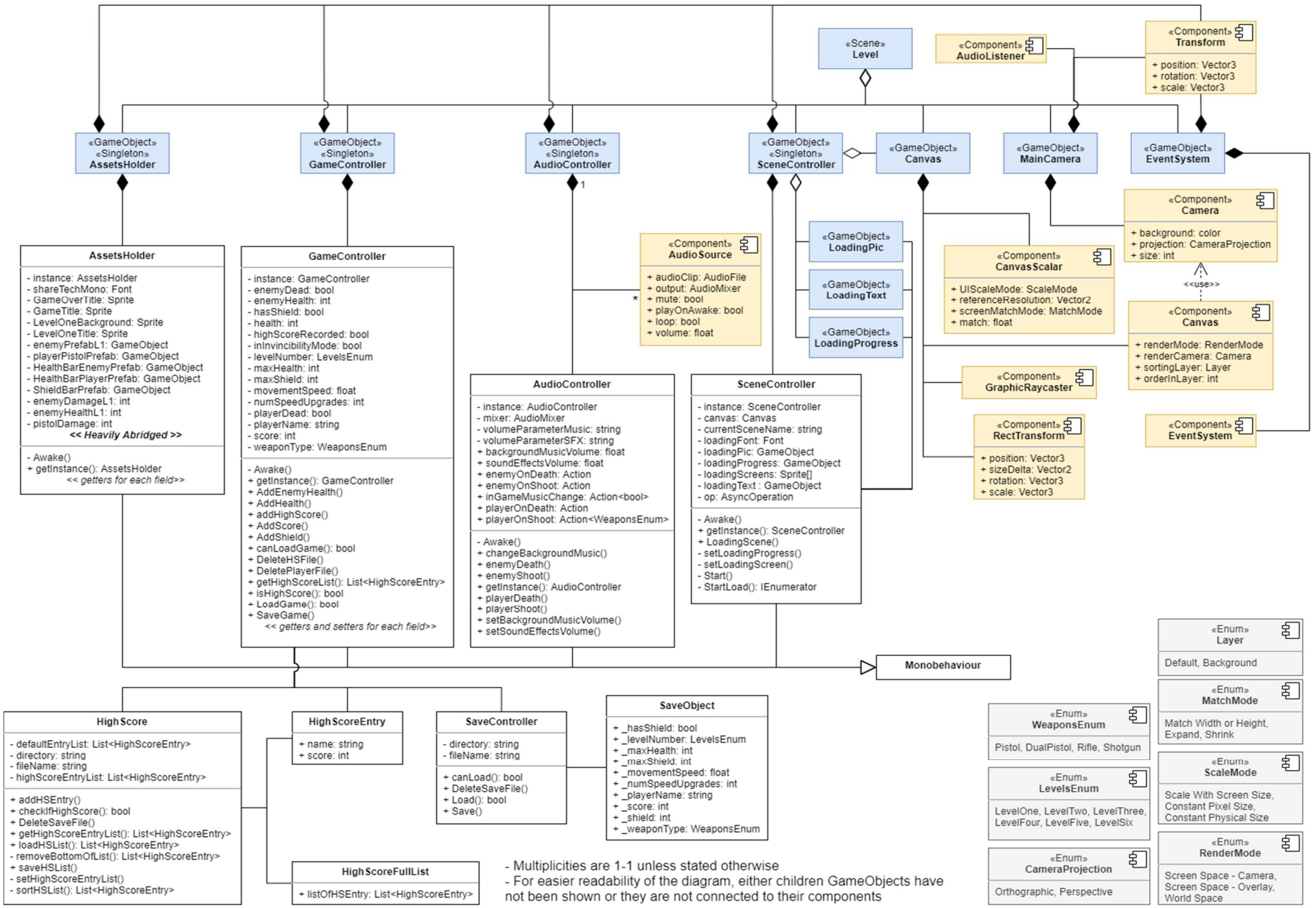


Figure 8: Class diagram of the level's scene base hierarchy.

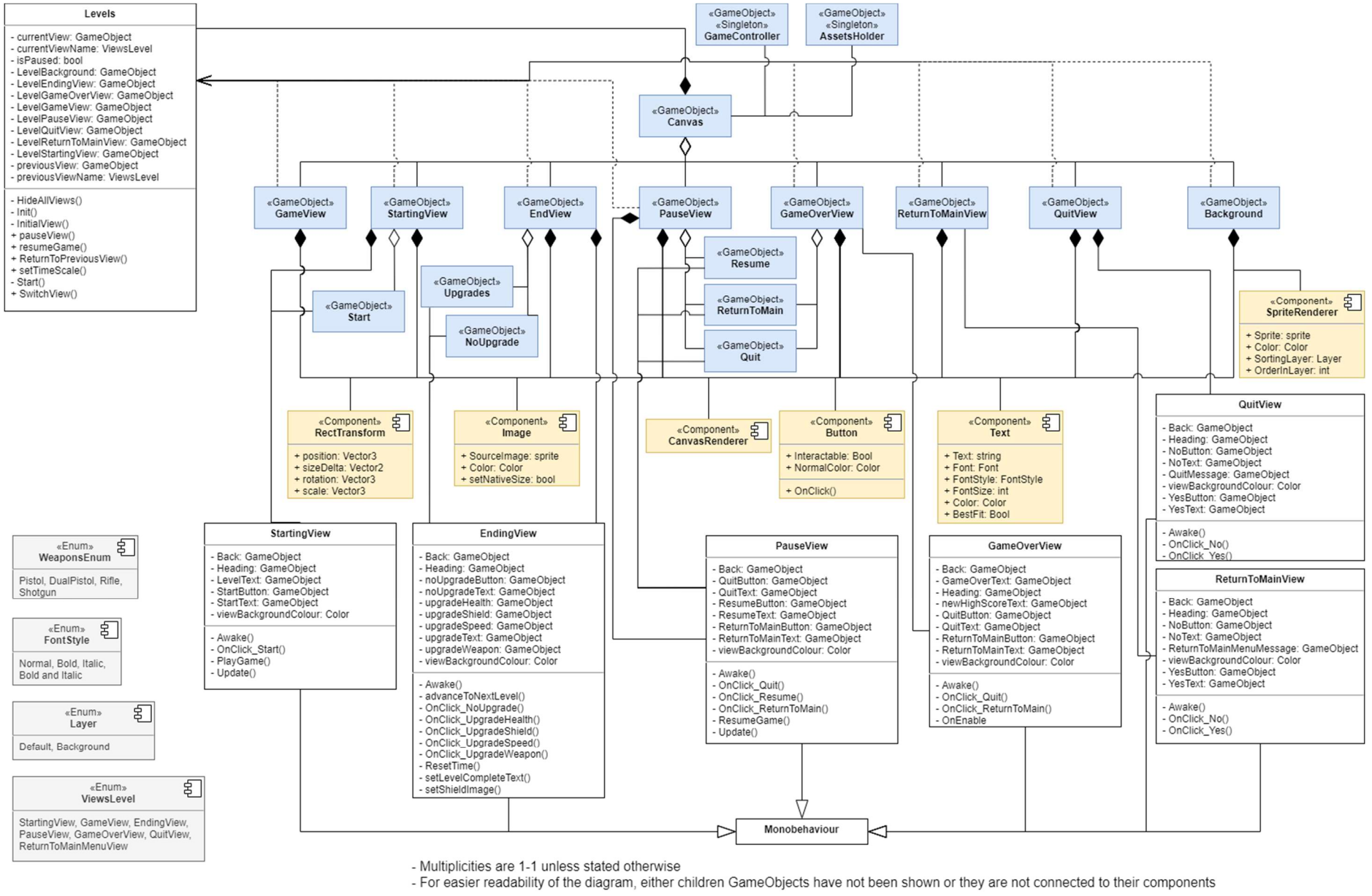


Figure 9: Class diagram of the level's main canvas and views (excluding the game view)

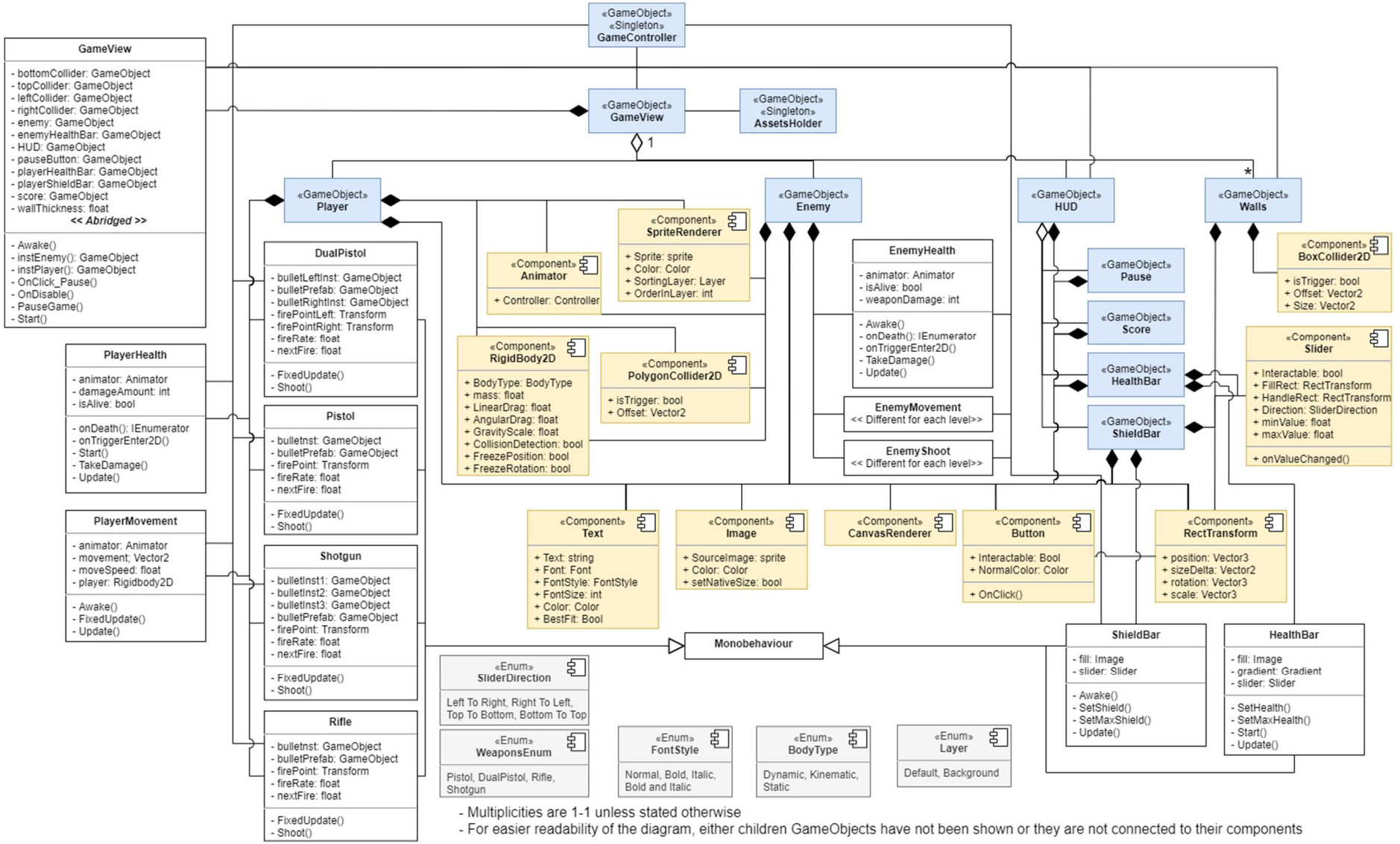


Figure 10: Class diagram of the level's game view

4.6. Project Management

There is a major issue within the gaming industry surrounding time management, as only 16% of projects are completed on time and on budget. There are a multitude of contributing factors, but mainly the adoption of poor methodology for software creation is to blame (Kanode and Haddad, 2009). As a result, making the right choice of development methodology is important in game development. The agile methodology was chosen for this project due to its iterative nature, which is required for game development as requirements often change throughout the project.

The iterative steps followed were a combination of tradition software development and game development. The requirements analysis, design, development and testing steps (Easterbrook *et al.*, 2008) were repeated by first seeing the requirements for a particular feature, developing it and then testing the feature. Testing ensured its functionalities worked as intended and did not negatively impact the performance of the previously implemented features. Implementing features in this manner helped modularise the functionalities of the system, increasing its flexibility and scalability.

At the concept stage, a project proposal was written regarding the plans for developing the game as well as a Gantt chart (Appendix B). The chart provided a guide for each stage of the development as well as milestones to be reached at different stages of the project. Weekly logs were submitted entailing the work completed that week, the challenges presented and the intended plan for the following week. Weekly meetings with the project supervisor also took place, informing them of the current progress and the milestones that had been reached. A live demonstration of the game was also presented to the project inspector on completion of the development stage of the project.

5.0. Solution Implementation

This section details the main features of the game, the game mechanics, the graphical elements, the programming of each feature and the controllers that were implemented in the making of COVID Destroyer.

5.1. Art Assets

Artwork in games refers to everything the user can see on their screen, including the backgrounds, menu items, heads-up display, and the look of the characters. The initial impression of the game from user's perspective is given by the visuals and the sounds on the title or story screens, before any gameplay takes place. The visuals set the tone and feel for the game, helping distinguish itself from others in the market (Rogers, 2014). Consistency in art style is essential for usability, so that users can fully immerse themselves into a world without feeling some objects are out of place. Therefore a pixelated art style was adopted throughout COVID Destroyer.

An abundance of free game art is available online. However trying to find assets to meet the exact requirements, from different artists with different styles, whilst trying to keep the design of the game consistent is challenging. Thus, most visual assets were created during the development process on PixelArt (Ware, 2021) and edited further on Gimp (Gimp Development Team, 1998). This allowed each element to be customised, to match the game style.

Figure 11: Player, enemy (adapted from (Ho *et al.*, 2020)*) and projectile art created for the game.*

Figure 11 shows the main player and enemy assets that were designed. Further artwork is shown throughout the report and in Appendix D.

5.2. Player

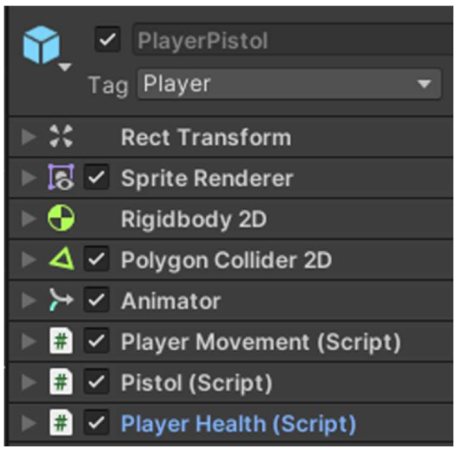


Figure 12: Player components

From the user's perspective the player is a simple image with movement animations. However, the player GameObject is complex and consists of several different components (Figure 12). RectTransform dictates the position, orientation, and size of the object within the world. The Sprite Renderer contains the sprite file, and allows modifications to be made, such as colour changes and the layer order. The Rigidbody2D and PolygonCollider2D components make the player a physical item in the world, allowing for interactions with other objects. The animator controls the visual state of the player (i.e. when the user wants the character to move right, the player changes state from idle to moving and is represented on screen by playing the animation). The movement, weapon and health components attached to the player were coded specifically for the game to carry out the required behaviours. The player was made into prefabs in order to instantiate it at run time into each scene's game view.

5.2.1. Player Characteristics

The characteristics of the player can be defined as the basic values that make up the inner workings of the object. These properties and their purposes are detailed below:

- Player Name: String value containing the name given to the character by the user before the start of the first level. If no name was entered, this defaults to 'Commander'
- Weapon Type: Enum, declaring the current weapon of the player
- Health: Integer value stating the player's current health value
- Max Health: Integer value stating the player's maximum possible health. This value can change depending on the upgrades taken by the user. The player starts on max health at the beginning of each level
- Shield: If the player has the shield, this declares the integer value of the current state of the shield. The maximum shield value is fixed
- Movement Speed: Float value stating how fast the player can move across the screen

All of these values are held within the Game Controller and are accessed either on initialisation of the GameObject or at runtime.

5.2.2. Player Movement

The PlayerMovement class enables the user to control the position of the player on screen using the movement keys ('A' or left arrow to move left, 'D' or right arrow to move right). References to the Rigidbody and Animator components are cached in the Awake() function for faster access and improved game performance, as calls to them can be computationally expensive especially within Update() (Unity Technologies, 2005b).

```
player.MovePosition(player.position + movement * moveSpeed * Time.fixedDeltaTime);
```

The function above is called within FixedUpdate() to keep in-step with the rate of the physics engine. The player here refers to the Rigidbody2D component and is managed by the physics engine. As shown in Figure 3 updates to physics occurs before processing user input. This function requires a Vector2 input and moves the body to the new position. It takes the current position of the body and adds to it a multiplier of the input (movement), the movement speed and the interval in seconds at which physics updates are performed (Time.fixedDeltaTime). The user input is collected within Update() and determines the movement direction of the object.

5.2.3. Player Weapons

There are four types of weapons the player can use dependent on the upgrade choices: pistol, dual pistols, rifle, and shotgun (Figure 11). Each have different mechanics, adding variation to the game. The pistol is the basic starting weapon that deals minimal damage per shot at a slow firing rate[7]. The dual pistol, as the name suggests, has the player wield two guns shooting two bullets each time the user fires. The rifle has a much faster rate of fire, with more damage per shot, and a faster bullet speed. The shotgun instantiates three bullets with spread on each fire but has a slower firing rate in comparison to the others. Each weapon is an upgrade to its predecessor in terms of damage per second. Table 1 shows the numerical values to these weapon qualities.

The implementation of the firing rate (see code snippet below) requires the built-in Time function. The player can only shoot, when two conditions are met: the correct user input is given, and the current time is greater than the next allowed time to fire (nextFire). When

[7] Firing rate represents how fast a weapon will shoot when the user fires. In the game this is set as a timer.

these conditions are satisfied, the bullets are instantiated, the shooting sound effect is played and nextFire is reset.

```
if(Input == fire && Time.time > nextFire) {
        nextFire = Time.time + fireRate;
        Shoot(); }
```

Table 1: Details of the weapon stats

Weapon Type	Shot Damage	Firing Rate (sec)	Bullet Speed	Damage Per Second (DPS)
Pistol	5	0.5	10	10.0
Dual Pistol	4 (x2)	0.6	10	13.3
Rifle	7	0.4	15	17.5
Shotgun	5 (x3)	0.7	8	21.4

5.2.4. Player Health

The PlayerHealth class governs the shield and health of the player. On instantiation of the player GameObject, the player's health value is set to maximum, and detects if a shield is present. If the player is surrounded by a shield, it must be fully depleted before any reduction in the player health occurs. The health class controls the damage the player takes by reducing the health and shield values only if hit by an enemy projectile. The amount of damage taken per hit is determined by the current level.

The instance of the GameController stores the values of the player's characteristics. The PlayerHealth class accesses the health and shield values from the GameController and manipulates them. The user sees a visual representation of the health and shield values on the Heads-Up-Display (HUD) via health/shield bars. These values are stored within the GameController to decouple the bars from the player. They only read the values from the GameController and have no knowledge about the player's other components. This does introduce an element of tight coupling to the GameController. However, modularity improvements were made, ensuring the player can exist without a health-bar and vice versa. The PlayerHealth class is also responsible for shield animations of the player.

5.2.5. Animations

The maintenance of a set of animations and associated animation transitions of a GameObject are maintained in the Animator Controller (Unity Technologies, 2005b). Unity provides a method to create animations or states the object can be in, and switch between these states dependent on game conditions using the controller. Figure 13 shows the Animator Controller for the player. The player starts in the idle state. If it has shield, the animation state changes to idle with shield, where a pink outline around the player is displayed (Figure 11). Through user input, the player transitions to the moving states contingent on the direction of motion and the speed. Upon player death (i.e. its health is reduced to 0), the final state showing the players death animation is achieved before the object is destroyed.

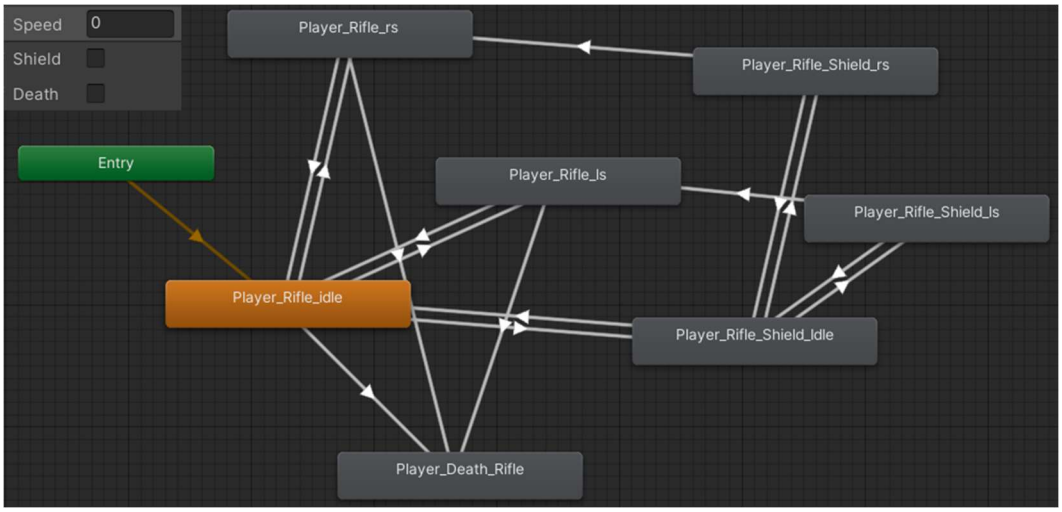


Figure 13: Player's Animator Controller, the different animation states, and the transition conditions

5.3. Enemy

The enemy prefabs have a similar structure to the player (Figure 12). The difference being the mechanics of the movement, shooting and health components. The enemy behaviour differs each level. Resultantly, separate prefabs with different scripts were created for the enemy.

5.3.1. Enemy Characteristics

The enemy behaviour differs from the player, in that no user input is required for the enemy to perform actions. As a result, most enemy characteristics are predetermined by the system in each level.

- Health: integer value stating the enemy's health, stored within the GameController

- Max Health: Integer value stating the enemy's maximum possible health for that level. This value differs each level and is stored in the AssetsHolder
- Damage: Integer value determining the enemy's damage per shot if the projectiles collide with the player. This value differs each level and is stored in the AssetsHolder
- Modes: Normal, Rage and Invincibility modes (discussed in the following section)

5.3.2. Enemy Types and Modes

Level difficulty in games is often overlooked. A common approach in increasing the difficulty as the player levels up, is to increase the enemy's health and damage output. Although this an effective approach, it lacks creativity from a design standpoint. Classic video games relied on different patterns for the increased challenge, requiring users to utilize memory and trial and error over strategy to advance further. However, there are other elements of gameplay that can be tuned or implemented to increase the difficulty from level to level (Boutros, 2008).

A number of different methods were employed to increase difficulty each level for COVID destroyer, one of which is changing the behaviour of the enemy. Alongside simply boosting the enemy health and damage, each level introduces a new mechanic for the user to combat.

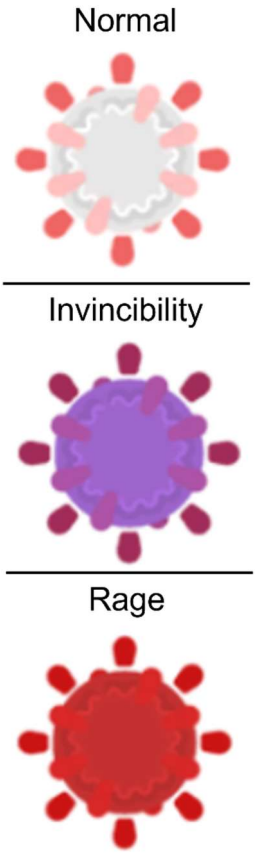


Figure 14: Enemy modes

In level one, the enemy possesses a health of 100 and shoots projectiles in a pattern, varying the angle of trajectory after each shot. This is accomplished using a Coroutine, which applies a delay between shots by yielding for a fraction of a second. The code below shows the algorithm used to accomplish this behaviour.

```
while (enemyIsAlive) {
        ShootAtAngle(counter * angleChange);
        if (!counterSwitch) {
                counter++;
                if (counter > counterMax) {
                        counterSwitch = true;
                        counter = counterMax; } }
        else {
                counter--;
                if (counter < counterMin) {
                        counterSwitch = false;
                        counter = counterMin; } }
        yield return delay; }
```

The counter number determines the path angle the projectile takes. The range of the angle needs to be limited to the player's range of motion, resulting in the need for counterMax and counterMin fields. The couterSwitch flag determines which direction the angle changes in (clockwise or anticlockwise).

Level two introduces basic enemy movement and a different shooting pattern. The COVID enemy moves side to side, changing direction on collision with the arena[8] walls. Projectiles are shot straight down from the enemy's current position. Its movement speed also increases as its health decreases, adding an additional element of difficulty within the same level.

Level three presents a new challenge due to the addition of enemy randomness. In previous levels, the opponent was akin to the enemies of classic games, where pattern recognition and basic gaming skill from the user was required to advance. The enemy fires down three projectiles at once with random times between shots, demanding the user to respond quicker to avoid taking damage. In terms of movement, the enemy GameObject chooses periodically, without bias, the new direction of motion (either stay in place, move left or right).

```
while (enemyIsAlive) {
	direction = randChangeDirection();
	enemy.velocity = direction * transform.right * speed;
	yield return delay; }
```

The code above shows by setting the direction to some integer between -1 and 1 inclusive, the enemy can be stationary (direction = 0), move left (direction = -1) or right (direction = 1). The movement is therefore unpredictable and prevents the user from simply learning a pattern and exploiting the behaviour to progress through the game.

Level four introduces two new game mechanics: targeted firing and 'Rage Mode'. Projectiles are now aimed at the player, requiring users to constantly dodge the incoming fire. Rage Mode fires targeted bullets at a much faster rate for a short period of time. Throughout the duration of this mode, the enemy is red (Figure 14), giving the user a visual clue as to the change in state of the enemy. The code snippet below illustrates the method used to switch between Normal and Rage modes.

[8] The arena refers to the space the player, enemy and bullets are contained within. They are surrounded by 4 invisible walls. Two on the top and bottom edges of the screen, and two pushed further in to the left and right of the screen.

```
while (enemyIsAlive) {
	isNormalMode = true;
	StartCoroutine( normalMode() );
	yield return delay;
	inNormalMode = false;
	yield return StartCoroutine( rageMode() ); }
```

The function normalMode() runs in a while loop on the condition isNormalMode is true. Thus, the property needs to be set to false to finish the Coroutine before initiating rageMode(). Both modes can be implemented at the same time removing the need for the Boolean flag, but this did not match the desired actions. The final statement in the code allows the current Coroutine to start the rageMode() Coroutine and yield until it is finished.

Level five's enemy replaces Rage mode with Invincibility mode, where it cannot take damage, shown by the enemy turning purple (Figure 14). In addition to this, the enemy gains the ability to dodge, by detecting incoming bullets and increasing its speed avoid the projectile. This is implemented through a CircleCast (as shown in the code below).

```
void FixedUpdate() {
	RaycastHit2D hit = Physics2D.CircleCast( gameObject.transform.position +
			offset, radius, direction );
	if ( hit.collider != null && hit.collider.tag == "PlayerBullet" &&
		Time.time > nextDodge ) {
			nextDodge = Time.time + dodgeRate;
			StartCoroutine( Dodge() ); } }
```

A circle is cast some distance away from the enemy towards the player. If the cast detects a collider and the object is a player bullet, the enemy undergoes its dodging method. When first applied, the enemy was free to dodge every frame a bullet was detected. However, after testing this feature, the enemy evaded most of the bullets presenting too difficult a challenge to overcome. Therefore, the dodge rate was introduced to apply a delay between dodges.

Level four employs the strategy of dealing as much damage as possible from the enemy to the player, whereas level five utilises the strategy to take as little damage as possible. Very different approaches must be taken by the user to advance through these levels. Level 6 further increases the difficulty by combining the previously mentioned features. The enemy switches between normal, rage and invincibility modes, increases speed as health depletes and has the ability to dodge bullets. Additionally, the enemy moves closer to the player as its

health depletes, adding a further hurdle for the user to overcome. Figure 15 illustrates the enemy activities within the final level.

5.3.3. Enemy Health

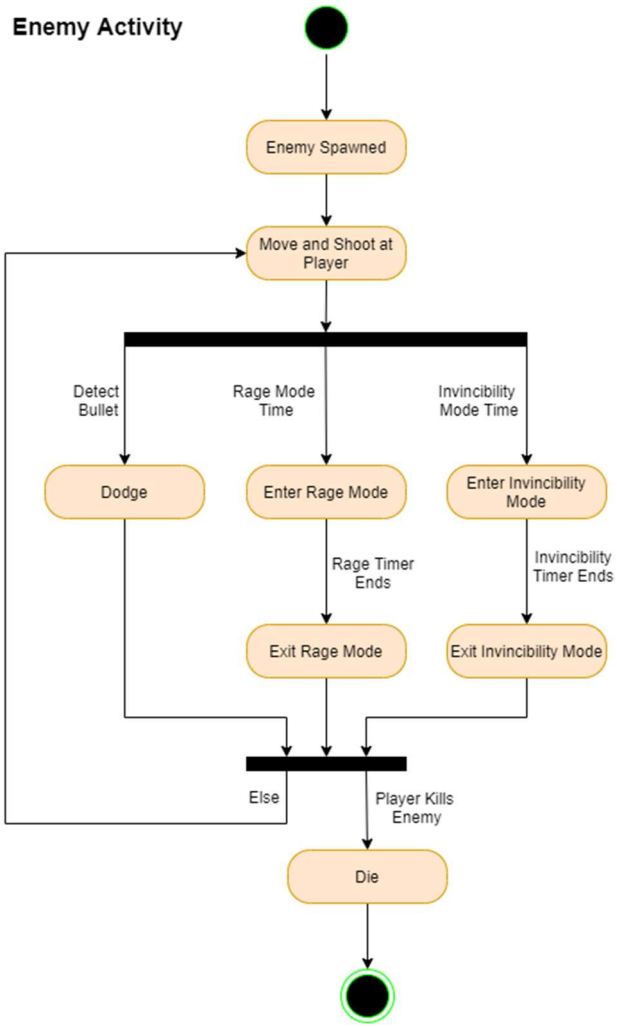


Figure 15: Enemy Activity Diagram on Level 6

The EnemyHealth class works similar to that of the PlayerHealth with a few exceptions. The enemy does not possess a shield, so any reference to it is removed. Furthermore, due to the enemy's invincibility mode, a check is performed when the player bullet collides with the enemy GameObject. If invincibility mode is currently active, no damage is taken, represented as the enemy's health remaining the same.

5.3.4. Animations

As the enemy mode changes, so does the enemy animation. The enemy can transition from normal state to invincibility, rage or death states. It cannot transition directly from rage mode to invincibility without first returning to the normal mode. Due to the nature of invincibility mode, no transition exists from this state to the death state as the enemy cannot take damage. Animations for each of these states were created on Unity. Figure 16 shows the enemy death animation.

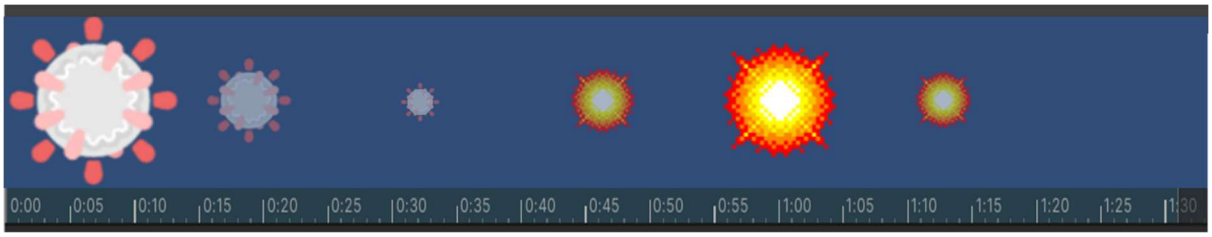


Figure 16: Enemy death animation

5.4. Projectiles and the Scoring System

The enemy and player both instantiate the projectile prefabs with intent to damage the other. Both contain a firing point from which the bullets spawn, at which point the script attached to the projectiles enact the behaviour. The code fragment below is run in the Start() method of the Fire class. The direction dictates whether the Rigidbody moves up the screen (for the player) or down (for the enemy). The bullet speed changes depending on the player's weapon choice (Table 1).

```
rbBullet.velocity = direction * transform.up * speed;
```

The enemy possesses two types of projectiles: untargeted for levels 1-3 and targeted from Levels 4-6. For the first three levels the projectiles work similarly to player's bullets, with the direction changing from 1 to -1. However, targeting the player is more complex as demonstrated in the code below.

```
player = GameObject.FindGameObjectWithTag("Player").transform;
target = (player.position – gameObject.transform.position).normalized * speed;
rbBullet.velocity = new Vector2(target.x, target.y);
```

First a reference to the player GameObject is captured the moment the projectile is instantiated. The target for the projectile is set by subtracting the position of the firing point from the referenced position of the player. The velocity of the bullet is then set by multiplying the target position by the speed. An attempt was also made in getting the bullet to follow the player even as it moved, by resetting the target on each Update() loop. However, this removed the player's opportunity to evade the attack, so was consequently removed.

The Fire class also contains the scoring system for the game. Each time a projectile collides with another object the OnTriggerEnter2D() method is called. Within this, the collided object's tag is found, and a score is calculated via the GameController. The scoring system rewards accuracy through adding 20 points for each direct hit and subtracting 1 point for each bullet that misses the enemy. Users also lose 3 points each time the player is hit by an enemy projectile.

5.5. Collision Detection

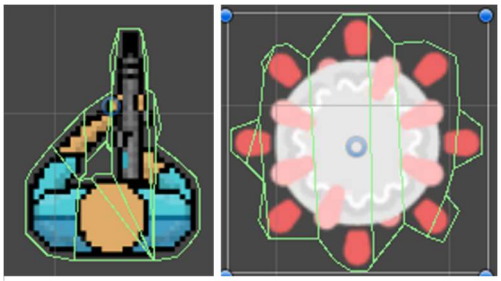


Figure 17: Player and Enemy polygon colliders shown as green outlines

A collider component is attached to GameObjects to reflect their structure and simulate their physics. The GameObject also requires a Rigidbody component as collisions are physics events. As a result, all in-game objects (i.e. player, enemy, bullets, walls) have both a Collider and Rigidbody. However, the type of collider component differs. Projectiles and walls use a BoxCollider2D component, whereas player and enemy GameObjects hold PolygonCollider2D components (Figure 17), the difference in these colliders lies with the shapes. A bullet is characterised as a rectangle. Although this is not an exact match to the shape of the projectile, in practice the behaviour is nearly identical due to the size of the asset. Characterising the bullet as a rectangle allows for additional performance benefits as box colliders are less computationally expensive in comparison to polygon colliders (Unity Technologies, 2005b).

Two methods could be employed to execute actions when a collision is detected: OnCollisionEnter2D() and OnTriggerEnter2D(). The bullets are set as triggers, allowing them to pass through other GameObjects (i.e. other bullets), therefore utilise the OnTriggerEnter2D method. Walls, players, and enemies all use OnCollisionEnter2D(), as collisions with these type of objects needs to block movement, for example a player colliding with a wall.

5.6. Health and Shield Bars

Figure 18: Health-Bar

The health and shield bars are a visual representation of the player's and enemy's health/shield. This was achieved using custom designed sliders. On loading a level scene, the health-bars are instantiated within the HUD. The HealthBar script, first reads the maximum health value of the player from the GameController within the Start() method and sets the slider value to its maximum. Within Update(), the health value is constantly read from the controller and displayed on screen. For improvements in usability, the fill of the bar has a gradient; as the health decreases, the colour changes to indicate this shift (Figure 18).

5.7. Controllers and Data Persistence

One of the biggest challenges in game development is data persistence. This entails preserving data between scenes and game executions. In Unity, when a new scene is loaded, all objects in the previous scene are destroyed, including the data contained within them (Unity Technologies, 2005b). In the majority of cases, data would need to be saved such as the player's characteristics. There are several methods of accomplishing this in Unity:

- PlayerPrefs: A very simple way to save attributes and their values using Unity's built-in functionality. The data is stored on the user's device, separate to the system and allows both scene and session persistence. However, only Float, Int and String values are supported. PlayerPrefs is commonly used for saving user setting preferences between game sessions (Zgeb, 2021).
- File Handling: Unity has built-in API (JsonUtility) for serialising and deserialising JSON data. The JSON data can then be saved into an external file. This allows for the creation of multiple save files. The file can be loaded on the load of the application or the new scene. However, this process is slower than other methods, reducing the performance of the game (Dickinson, 2017).
- Static Class: A single static database-type class, which cannot be attached to an object. It holds all the data and can be globally accessed enabling data persistence between scenes.
- Singleton Pattern: Holds some similarities with the static class but can be attached to a GameObject. DontDestroyOnLoad() is assigned to the object holding the instance, preserving the object between scenes.

For COVID Destroyer, two of these methods were utilised. The Singleton Pattern for data preservation between scenes and file handling for data between game sessions. All three controllers shown in Figure 5 make use of the Singleton pattern. The differentiation between the controllers is reliant on the type of data they hold and the functions they manage.

The AssetHolder class maintains all the constant data (such as the enemies health at the start of each level, the amount of damage each weapon does and all the art assets). Although it is resource intensive to hold a reference to all art in one class as opposed to loading the resources within the class when and where needed, it allows for better performance at run time through getting the instance of the AssetHolder. A Singleton was not necessary in this

case as a static class would have sufficed. However implementing the pattern allowed the component to be attached to a GameObject. As a result, the Unity Editor was used to drag and drop the assets into the AssetHolder GameObject.

5.7.1. Game Controller

The GameController class contains player characteristics (Section 5.2.1), variable enemy characteristics, user's score, current level, Boolean flags that dictate game state (playerDead, enemyDead fields), methods to save and load the game, add score method, and getters and setters for each of the values. It works as a global access point for all other classes, to retrieve and change data through GameController.getInstance(). The design choice was made to decouple functionalities from each other, resulting in loose coupling between other classes, but tight coupling between these classes and the GameController. This improved the modifiability and flexibility of the system, as new features were added to the game without effecting other functionalities.

The controller classes can only have one instance of itself at any given time. On loading into a new scene that also contains the controller prefab, the controller in the new scene detects if a current instance exists and if so, destroys itself. This allows the game to start at any level, preventing the need to begin each session at the title scene. Although from the user's perspective the application always leads them to the title screen before any levels, this method allowed testing of new features in later levels much easier when developing the game.

When a new instance of the GameController is created, the data within it starts with default values (e.g. player's starting max health is 100). As the user progresses through the game and gains upgrades, the data changes and is saved on the completion of each level. Upon player death, or completion of all levels, this instance needs to be reset to its default values in order to start a new game. Due to the previously mentioned method of having a controller prefab on each scene, this instance of the GameController can simply be destroyed before returning to the title scene where a new instance will be created with the initial values.

5.7.2. Save Controller

The saving and loading of game states are achieved by the GameController holding methods that delegate to the corresponding methods in the static SaveController class. This reduces

coupling between other classes and the SaveController. The Save() method is called the moment the user chooses an upgrade at the end of the level and before the new scene is loaded. Within this method, data that needs to be stored is retrieved from the GameController and placed into a new serializable SaveObject class. This SaveObject is converted into JSON format via the JsonUtility package, before being written into a text file. The Load() method works in reverse, and is initiated from the title screen when the user presses the 'Load Game' button. It first checks to see if the player file exists, before reading the text file and converting the JSON string into a SaveObject. This data is loaded into the GameController if the user chooses to load a game.

5.7.3. Scene Controller

Where the GameController manages the game state, the SceneController manages the loading of new scenes. Unity allows for asynchronous loading of scenes, meaning the new scene is loaded in the background over several frames rather than all at once, which could cause the game to freeze. The SceneController employs this as seen in the code below[9].

Figure 19: Example of a loading screen

```
IEnumerator LoadScene( string sceneName ) {
        selectLoadingScreen()
        canvas.gameObject.setActive( true );
        operation = SceneManager.LoadSceneAsync(sceneName);
        while( !operation.isDone ) {
                setLoadingProgress ( operation.progress ); }
        yield return delay;
        canvas.gameObject.SetActive( false ); }
```

The loading image is dependent of the state of the game (see Appendix D). The different loading screens enhance the story aspect of the game by highlighting the areas of the world that have been cleared from the virus by the player. The correct loading screen must first be chosen before activating the canvas that displays the image. The LoadSceneAsync() is seen as

[9] The code is truncated to show the main functionality of the Coroutine.

an operation, which allows the progress to be tracked. This progress is presented as a percentage to the user (Figure 19) to confirm the game is running. Once the operation is complete, the loading screen canvas is deactivated, presenting the new scene to the user.

5.7.4. Audio Controller

An important role is played by the sound design and music in video games. They build tension, add story and give feedback to the user about the current game events (Scarratt, 2018). The AudioController, as the name suggests, manages the sounds of the game. From the title screen onwards, a constant background music plays[10], enhancing the user experience. The music changes when the user is in-game[11], where different sound effects are also played depending on the actions carried out (such as: player and enemy shooting and death)[12]. The sound for each weapon differs, giving further feedback to the user about the weapon chosen. The AudioController houses an AudioSource component for each of these sounds.

The Observer Pattern was implemented to play the sounds. When an event occurs (e.g. the enemy dies), the subject broadcasts this change to observers listening for the event, which in turn carries out the desired functions, in this case plays the enemy death sound.

The AudioController also handles the volume. A mixer with two groups – one for background music and the other for sound effects (SFX) – allows the user to control the volumes of different types of sounds separately.

5.8. Story and Title Scene

Upon opening the application, the user sees the story screen, where the animated text informs of the current situation concerning corona virus within the game. It presents the objective they are trying to achieve, which is to save the world by defeating COVID. The story screen (Figure 20) instantly engages the user through storytelling, setting the goal for the game, visuals, and interactivity. By pressing 'enter' the user transitions to the title scene.

[10] https://soundimage.org/looping-music/
[11] https://mixkit.co/free-sound-effects/game/
[12] https://freesound.org/people/chrisw92/sounds/179673/
https://freesound.org/people/MATRIXXX_/sounds/441373/
https://freesound.org/people/HappyParakeet/sounds/398068/
https://freesound.org/people/Jofae/sounds/363698/
https://audiosoundclips.com/8-bit-explosion-blast-sound-effects-sfx/

Figure 20: Story screen (top-left), Main title view (top-right), Settings view (bottom-left) and High Scores view (bottom-right)

The title scene consists of many views, which are activated when the user clicks on various buttons. For example, when the user clicks the settings button of the main title view, the system will display the settings view (Figure 20). All other title screen views can be seen in Appendix D.

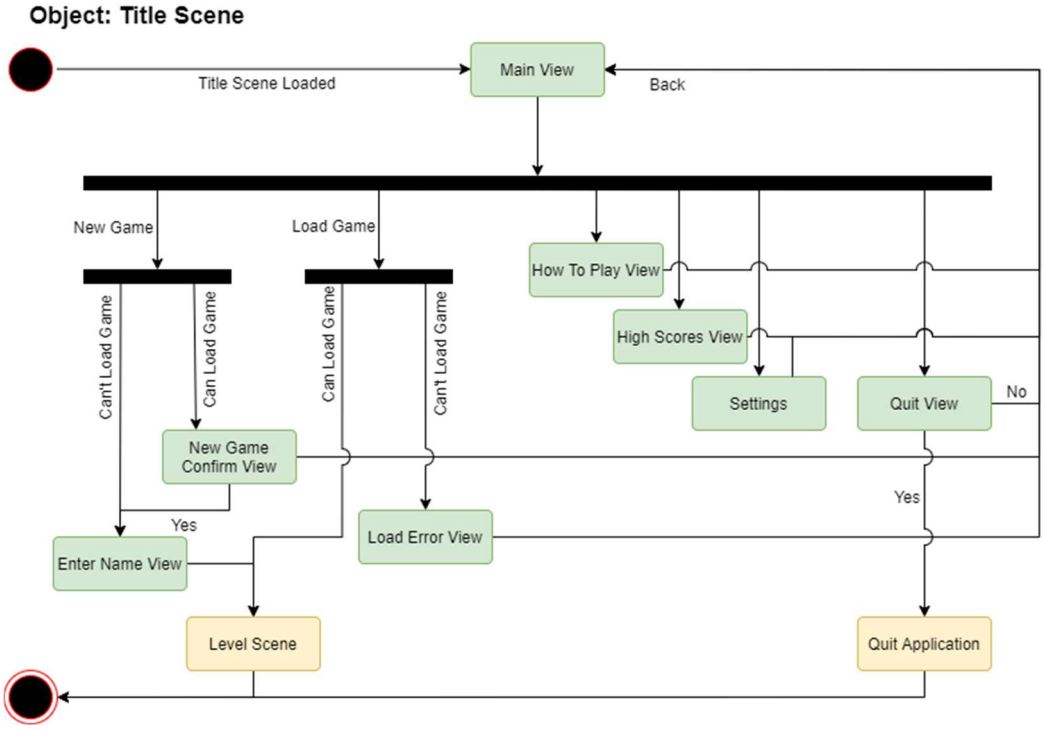


Figure 21: UML State diagram of the views on the title screen

Usability was found to be a key attribute in the design overview, ensuring the user is capable of navigating through the system with ease. All buttons are clearly labelled, fonts and colour schemes are consistent throughout, and the system clearly illustrates the actions that will occur by confirming user choices (Bernard, 2003). Figure 21 demonstrates the transitions between different views within the title scene through a state diagram.

5.8.1. Settings

Modern video games often have a plethora of settings that can be manipulated by the user, including the controls, graphics, sound, resolution, display and many more. Being a smaller game, COVID Destroyer did not require an extensive collection of settings. The user can adjust the volume of the sounds via sliders and change the resolution of the game with toggles. All resolutions have a 16:9 aspect ratio. A graphics quality setting was also implemented, but due to the pixelated art style, this had little to no effect on the visual or performance aspect of the game and was therefore removed.

5.8.2. High Scores

Scores indicate how well the user has performed in a game, giving them a goal to reach and a sense of achievement when the goal is met (Harteveld and Sutherland, 2015). In COVID Destroyer, the top ten scores on the system are recorded and displayed in the High Scores view (Figure 20). The HighScore class saves these scores in JSON format into a separate text file, allowing them to be loaded on following game sessions.

Before the first instance of playing the game on a device the high score file is yet to exist. The HighScore class creates this file and populates it with default player name and score values. On each attempt of the game, user's score entries are added to the list, in descending order. An entry is added only when the player dies or has completed all six levels. The player's name and score are sent to the HighScore class, where the entry is added to the list. A Bubble Sorting algorithm is executed and any entries below the tenth entry are deleted before the list is saved. The saving and loading follow a similar process discussed in Section 5.7.2.

5.9. Level Scenes

All levels are all made of the same number of views (Figure 7B), the difference being the content within them. The Levels class attached to the main canvas of the scenes creates these

views as GameObjects, attaches individual scripts to these views, and controls their activation state[13] based on user action by using switch cases. Every level begins with the Starting view.

The Starting view continues the story element of the game, by displaying text that informs them about the current pandemic within that continent (i.e. the level). It also contains information about the enemy name and the type of movement and attack it possesses to prepare the user for the gameplay.

5.9.1. Game View

The Game view (Figure 22) consists of the player, enemy and the HUD, which contains the health-bars, score and pause button. As discussed previously, the health-bars, enemy and player are all prefabs and thus are instantiated once the scene has loaded. The text of the score updates each frame along with the health/shield-bar texts. The Game view transitions to the Pause view via user input, and to the other views through a change in game state; either the player or enemy dying (Figure 24).

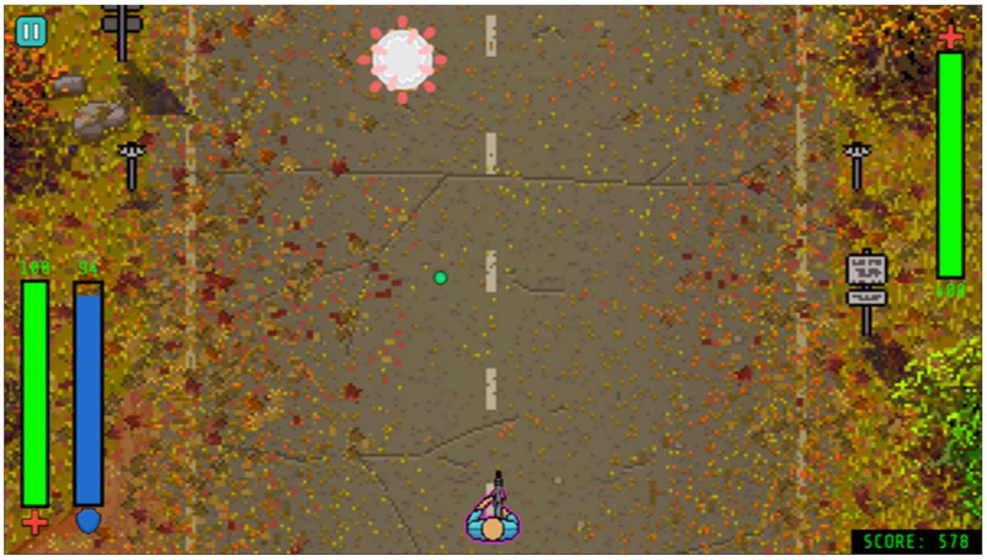


Figure 22: Game view on Level 4

5.9.2. Pausing

Most operations that require user action and GameObject movement are time based. Unity provides methods to manipulate the time function to either speed up or pause all time-based operations including physics and animations. When the user pauses the game, the Time.timeScale is set to 0, and reset back to 1 when the game is resumed. The Pause menu allows the user to return to the main menu or quit the game.

[13] A view is active if its GameObject is enabled and is shown on the user's screen

5.9.3. Upgrades

As the enemy increases in difficulty each level, the player needs new strengths to overcome these new challenges. This comes in the form of upgrades. On completing each level the user chooses one of the following (Figure 23):

- Weapon Upgrade: Pistol, Dual Pistols, Rifle and Shotgun
- Speed Boost: Increases the movement speed of the player
- Health Upgrade: Increases the maximum health of the player
- Shield: Player gains a shield. If they already have a shield, the shield value is reset to maximum upon choosing the upgrade again

The upgrading game mechanic presents the opportunity to apply different strategies to defeat the enemy. This also increases the game's replayability as different upgrade paths can be explored and with the 'No-Upgrade' option adding further difficulty to the game.

Figure 23: Upgrades that can be chosen by the user to increase the player's abilities

5.9.4. State and Activity Diagrams

The levels scene possesses many states, where various conditions, either user or game status driven, allows transitions to the next state. The state diagram in Figure 24 is an illustration of the views transitioning (see Appendix D for the views). The activity diagram (Figure 25) demonstrates via swim-lanes, the user and game activities throughout the level.

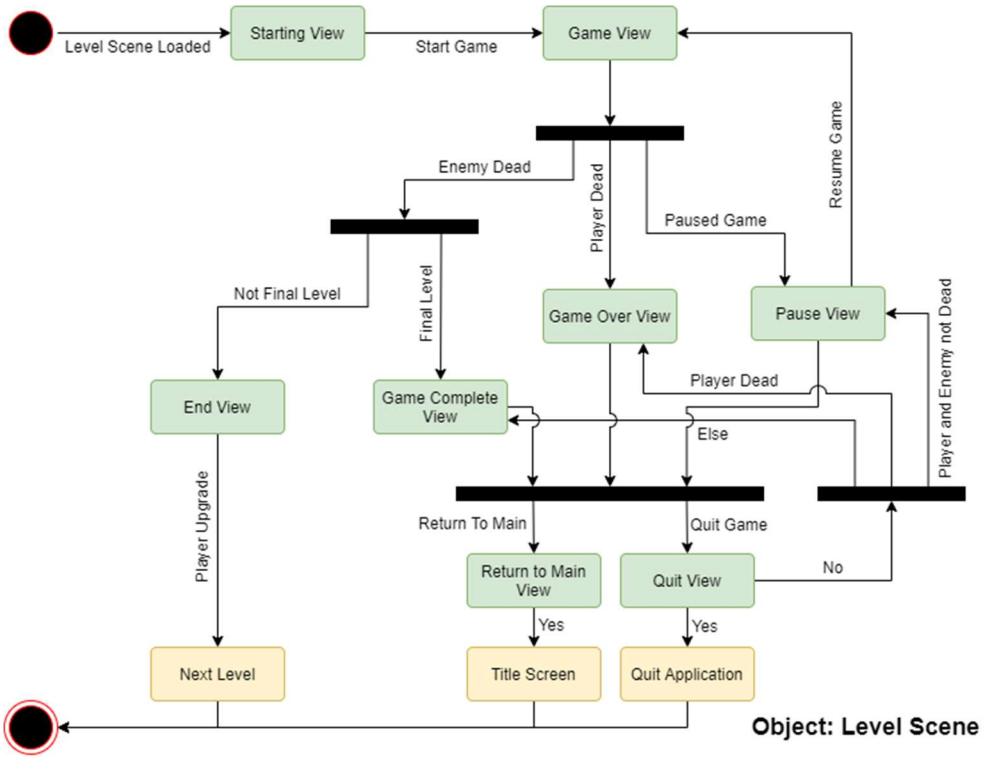


Figure 24: UML State diagram of the Level scenes

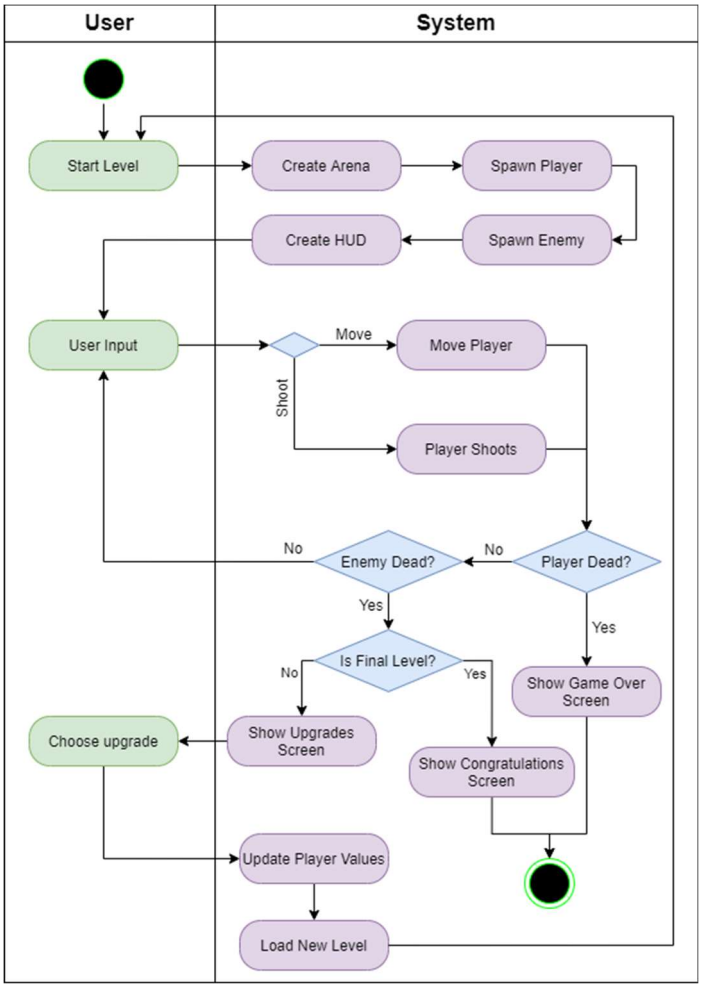


Figure 25: UML Activity diagram of the levels

5.10. Deployment

The system was built on the Unity Engine using a Windows device. One of the main strengths of the Unity Engine is the ease of cross-platform porting. As a result, creating three separate builds for Windows, macOS and Linux only required a few build settings to change. MacOS computer's often use a different aspect ratio in comparison to Windows, therefore the aspect ratio for the builds were locked to 16:9, where letterboxing[14] would occur if different screen resolutions are used. The use of player logs was enabled only for the development build as logging on Unity decreases the system performance.

[14] Letterboxing refers to the black bars that appear on the edges of the screen when an aspect ratio is preserved instead of stretching to the size of the screen.

6.0. Testing and Evaluation

6.1. Testing

Quality assurance (QA) is a critical part of game development. Ideally, a game would be tested meticulously by continuous playthroughs, trying every possible combination of events, in every location of the game. The central issue regarding this method is the time required to complete the testing (Bethke, 2003). It is not certain it would be possible to play through a game in its entirety, especially in modern video games where the world space can get incredibly large. The game industry is yet to find an all-purpose testing method for games, where instead companies use their own QA process (Bethke, 2003). To test the game throughout the development process, three methods were implemented: black-box functionality testing, automated testing, and user testing.

Black-box testing entails the tester having no knowledge of the source code of the system; thus all testing is done from outside the application. The tester carries out steps and records the outcome. If the outcome is not as predicted, it is recorded as a bug and fixed on the next update of the game (Schultz and Bryant, 2016). Appendix E shows a list of test cases for such testing. Automated testing helps check small portions of the code function as expected. The Unity Test Framework allows testing in both Edit and Play mode (Unity Technologies, 2005b). User testing, as the name suggests, involves numerous users playing the game and reporting on any performance issues or bugs. The system was also tested throughout the development process using logs and print statements within the Unity editor. Due to the employment of the Agile methodology throughout the proect, a constant cycle of design, develop and test new features were carried out.

Automated testing of every functionality of the game, with every type of input is incredibly time consuming. Additionally, due to the nature of games being visual software and Unity's inheritance of MonoBehaviour on all classes attached to GameObjects, traditional unit testing (Unity refers to as Edit mode) proved to be difficult. Therefore the decision was made to automate the testing of some of the main functionalities.

Edit mode testing runs in the editor, allowing testing for functions that do not require physics components. The GameController class contains various methods which can be tested

without any visuals and is a vital cmponent in controlling the game. The static SaveController class and the HighScore class do not extend MonoBehaviour, as a result were also tested through Edit mode. The methods within these classes were tested with various inputs ensuring outcomes were as intended. The creation and update of text files was also tested to ensure data persistence between game sessions. The completed checklist of these tests can be seen in Appendix E.

Play mode testing runs in the game. It is a much slower process, as it requires scenes to be loaded. The tests themselves are much harder to write as it is often difficult to emulate the game and check for the expected results. Using Play mode, the player's movement was tested by using the NSubstitute library[15] to mock user inputs to test the players movement. Further Play mode testing involved testing for bullet movement, collisions with objects with different tags and bullet instantiation through mocking user input. Appendix E shows the list of Play mode tests carried out.

User testing was fulfilled together with evaluation via a questionnaire, the results of which are discussed in Section 6.2. Through user testing, some bugs and quick improvements were found early on. A reported crash occurred when the user tried to return to the Title scene. It was found that the game had been paused just before the player died, resulting in the engines Time.timeScale being set to 0 when the user was on the Game Over screen. As a result, the engine's time scale was never returned to 1, making it appear as if the game had crashed. This was resolved simply by ensuring the time scale was always 1 before transitioning between scenes.

User's suggested improvements that helped the usability and balance of the upgrades. The health and speed upgrades were deemed not to be as effective. Furthermore, there were some confusions about the amount of health the enemy has in each level as the health bar size remained the same. As a result, the health and speed upgrades were improved by increasing the amount of health gained and the movement speed each time the upgrade weas chosen. Text was added near the health bars, indicating the numerical value of the player and enemy health's. These improvements were quick to implement and were added to the newest build.

[15] https://nsubstitute.github.io/

6.2. User Evaluation and Feedback

A user evaluation questionnaire was distributed amongst a case study of 10 users of COVID Destroyer. The questions prompted the users to evaluate the usability and performance of the system, as well as report any bugs or crashes. Overall, COVID Destroyer achieved an average user rating of 7.3, with only one user reporting any issues in performance. This issue is addressed in the testing section. The questions and results of the questionnaire are found in Appendix F.

The game was played on the required operating systems it was built for by multiple users (Windows, Linux and macOS). The user's played the game for an average of 2 hours, and most claimed they would play again. In the design objectives, high priority was placed on the usability of the system, ensuring the controls were simple to understand and the game menus were easy to navigate. All 10 participants agreed the game was intuitive and easy to use.

The game's story had mixed opinions, with 60% of user's agreeing COVID Destroyer has a good story. However, this did not seem to hinder user satisfaction as 80% of users agreed on the game being enjoyable. The survey also showed that all upgrades were explored extensively, and that half the users reached the final level. This suggests a good balance between the player and enemy abilities, whilst still presenting a challenge for users.

Users were prompted to suggest improvements to the game. Some suggested more levels and upgrade options, other's wanted improvements to the currently available upgrades which were implemented in the final build of the game. Other suggestions, such as an indication the player gets hit beyond the health-bar going down, would take longer to implement and are regarded as future work. As a whole, feedback for the game from the users was positive, holding many possible avenues of the expansion of the game.

7.0. Discussion

The project succeeded in its overall objectives of designing and developing a game utilising outstanding software and game development principles. The design objectives set out the attributes the system needed to achieve to be considered a successful product. The usability is difficult to judge as challenging the user is one of the objectives of the game. However, based on user survey's the system is easy to use and navigate, as well as being intuitive to understand in terms of the game's objective of controlling the player to defeat the virus. These characteristics demonstrate the usability of the system.

The performance and modifiability were also key attributes in developing the game. Often these attributes oppose one another, as systems designed for absolute performance are not modifiable (Lundberg *et al.*, 1999). However, an equilibrium was reached when designing the architecture of COVID Destroyer. Through the use of composition over inheritance, the system became more flexible and modular, being able to add features into the game without tight coupling between multiple classes. Through the use of singleton, component and observer patterns, in addition to minimising functions executed within Update(), the performance of the system was greatly increased. Besides the one user, no instances or reports of performance deficiencies were found.

Some users suggested more levels and upgrades, indicating the system could be expanded into a larger game or franchise. Implementing interfaces, would have allowed for better scalability of the system. However, adding this layer of abstraction for flexibility, adds complexity to develop, debug and maintain (Nystrom, 2014). This would take longer than the time allocated for this project.

Testability was overlooked in part during the design process. Extensive automated testing could not be accomplished partly due to the difficulties in developing these tests in Unity, and partly due to the code being written without the intent of automated testing. A test-driven development technique could be employed, where requirements are converted to test cases before the software is developed. However, this approach is often time intensive and due to the time constraints of this project, was not feasible.

Security was given lesser importance as it was deemed to hold a lesser relevance in comparison to other factors. However, some security measures could have been implemented especially within the external files in which data is written into by encrypting the player save files. Due to the size of the game, this was initially deemed as excessive. Security would need to be employed if a global high score were to be implemented, by storing the scores on a database.

7.1. Future Work

This project holds the potential for many pathways of future expansion. Further development of the game by adding more levels and upgrades were suggested by the users. Further upgrade could include a laser gun, which shoots a beam of light dealing more damage overtime while the laser remains in contact with the enemy. This would present a new mechanic to the game through a difference in weapon type. An inventory system could also be introduced for the player allowing the user to collect weapons and switch between them or earn special perks through in-game achievements. New enemy mechanics could also be introduced, by having hazard areas which appear randomly and deal damage to the user if they are within the hazard area. Ideas for expanding the game in the creative sense are endless.

High scores in games often create competition. Currently, high scores are saved onto the local machine. The implementation of a global scoreboard, by storing scores on a database, would allow users to compare themselves against others around the world, adding a further element of competition between users. Furthermore, a two-player mode with one controlling the player and the other the enemy, would add replayability value.

The game was designed as a desktop application. However, due to Unity's great cross-platform portability, minimal changes are needed to deploy the game as a mobile application. Users would control the player through an on-screen joystick or tilting the phone, tapping the screen to shoot. This would allow for COVID Destroyer to be available to a wider range of users.

8.0. Conclusion

This report has detailed the design and development of COVID Destroyer; A 2D game built on the Unity Engine. The story for the game revolves around the user defeating the corona virus on each level, eventually saving the world by completing all the levels. COVID Destroyer has 6 levels, each challenging the user by implementing a different enemy mechanic. The user is given choices on the completion of each level to upgrade the player, enabling different strategies based on the upgrades taken. The game was designed ensuring usability, performance, and modifiability, therefore implemented the Entity-Component architecture, choosing composition over inheritance for a more modular system.

The initial background research explored traditional software engineering projects and game development followed similar developmental procedures. The project specification was written alongside the requirements of the game, to detail the functionalities the game must have. Throughout the development of the system, good software engineering and game design practices were followed to an extremely high degree.

The game was built as a desktop application for Windows, macOS and Linux. The application was tested partly through automated testing within the Unity Framework, and through play testing. An evaluation of the game by a group of users found the game to be enjoyable striking the right balance between challenging and entertaining. The entirety of the software life cycle from initial conception through to deployment was undertaken; the end product being COVID Destroyer.

References

2K Games (2007) 'BioShock'. 2K Games, Feral Interactive, Take-Two Interactive, D3 Publisher.

Aleem, S., Capretz, L. F. and Ahmed, F. (2016) 'Game development software engineering process life cycle: a systematic review', *Journal of Software Engineering Research and Development*, 4(1), p. 6. doi: 10.1186/s40411-016-0032-7.

Audacity (2000) 'Audacity (3.0.4)'. Available at: https://www.audacityteam.org/.

Barone, E. (2016) 'Stardew Valley'. Chucklfish, Fangamer, 505 Games.

Bass, L., Clements, P. and Kazman, R. (2003) *Software architecture in practice*. Addison-Wesley Professional.

Beat The Bomb (2020) 'Fauci's Revenge'. Available at: https://beatthebomb.com/beatthevirus/.

Ben-Zahia, M. A. and Jaluta, I. (2014) 'Criteria for selecting software development models', in *2014 Global Summit on Computer Information Technology (GSCIT)*, pp. 1–6. doi: 10.1109/GSCIT.2014.6970099.

Bernard, M. (2003) 'Criteria for optimal web design (designing for usability)', *Retrieved on April*, 13, p. 2005.

Bethke, E. (2003) *Game development and production*. Wordware Publishing, Inc.

Boutros, D. (2008) 'Difficulty is Difficult: Designing for Hard Modes in Games', *gamedeveloper.com*. Available at: https://www.gamedeveloper.com/design/difficulty-is-difficult-designing-for-hard-modes-in-games.

Cook, M. (2020) 'Software Engineering For Automated Game Design', in *2020 IEEE Conference on Games (CoG)*, pp. 487–494. doi: 10.1109/CoG47356.2020.9231750.

Crytek (2002) 'CryEngine'.

Dar, R. (2021) 'Top 7 Gaming Engines You Should Consider for 2021', *Incredibuild*. Available at: https://www.incredibuild.com/blog/top-7-gaming-engines-you-should-consider-for-2020.

Dealessandri, M. (2020) 'What is the best game engine: is CryEngine right for you?', *gamesindustry.biz*. Available at: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-cryengine-the-right-game-engine-for-you.

Dickinson, C. (2017) *Unity 2017 Game Optimization: Optimize All Aspects of Unity Performance*. Packt Publishing Ltd.

Dillet, R. (2018) 'Unity CEO says half of all games are built on Unity', *techcrunch*. Available at: https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/.

Easterbrook, S. *et al.* (2008) *Guide to Advanced Empirical Software Engineering*. London: Springer. doi: https://doi.org/10.1007/978-1-84800-044-5_11.

Edwards, G., Li, H. and Wang, B. (2015) 'BIM based collaborative and interactive design process using computer game engine for general end-users', *Visualization in Engineering*, 3. doi: 10.1186/s40327-015-0018-2.

Epic Games (1998) 'Unreal Engine'. Cary, North Carolina.

Gamma, E. *et al.* (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education (Addison-Wesley Professional Computing Series). Available at: https://books.google.co.uk/books?id=6oHuKQe3TjQC.

Gimp Development Team (1998) 'Gimp'. Available at: https://www.gimp.org/.

Goldstone, W. (2009) *Unity game development essentials*. Packt Publishing Ltd.

GorillazXD (2016) *Fallout Screen*, *DeviantArt*. Available at: https://www.deviantart.com/gorillazxd/art/Fallout-Screen-624115776.

Harteveld, C. and Sutherland, S. C. (2015) 'The Goal of Scoring: Exploring the Role of Game Performance in Educational Games', in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, pp. 2235–2244. Available at: https://doi.org/10.1145/2702123.2702606.

Hay, M. (2020) 'Coronavirus-themed video games are everywhere', *Mic*, October. Available at: https://www.mic.com/p/coronavirus-themed-video-games-are-everywhere-40.

Heubl, B. (2020) 'Long-forgotten games from the past are making a comeback for a generation of millennials

under lockdown.', *Engineering and Technology*. Available at: https://eandt.theiet.org/content/articles/2020/05/retro-gaming-boom-during-lockdown/.

Ho, D., Kostman, M. P. and Steinberg, P. (2020) *Free COVID-19 (SARS-CoV-2) Illustrations*, *Innovative Genomics Institute*. Available at: https://innovativegenomics.org/free-covid-19-illustrations/#c-section-3.

IBM (2019) *Software development*, *IBM*. Available at: https://www.ibm.com/topics/software-development.

ISO/IEC 12207 (2008) 'System and Software Engineering - Software life cycle processes'. ISO. Available at: https://www.iso.org/standard/43447.html.

ITChronicals (2018) *What is Software Development*. Available at: https://itchronicles.com/what-is-software-development/.

Kanode, C. M. and Haddad, H. M. (2009) 'Software Engineering Challenges in Game Development', in *2009 Sixth International Conference on Information Technology: New Generations*, pp. 260–265. doi: 10.1109/ITNG.2009.74.

Kasurinen, J. (2016) 'Games as Software: Similarities and Differences between the Implementation Projects', in *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*. New York, NY, USA: Association for Computing Machinery (CompSysTech '16), pp. 33–40. doi: 10.1145/2983468.2983501.

Linietsky, J. and Manzur, A. (2014) 'Godot'. Open Source under MIT License.

De Lope, R. P. and Medina-Medina, N. (2016) 'Using UML to Model Educational Games', in *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pp. 1–4. doi: 10.1109/VS-GAMES.2016.7590373.

López-Cabarcos, M. Á., Ribeiro-Soriano, D. and Piñeiro-Chousa, J. (2020) 'All that glitters is not gold. The rise of gaming in the COVID-19 pandemic', *Journal of Innovation & Knowledge*, 5(4), pp. 289–296. doi: https://doi.org/10.1016/j.jik.2020.10.004.

Lundberg, L. *et al.* (1999) 'Quality attributes in software architecture design', in *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*, pp. 353–362.

Martin, R. C. (2000) 'Design principles and design patterns', *Object Mentor*, 1(34), p. 597.

Mechner, J. (1989) 'Prince of Persia'. Broderbund.

Microsoft (2015) 'Visual Studio Code'. Available at: https://code.visualstudio.com/.

Namco Bandai Games (1980) 'PAC-MAN'. Namco Bandai Games Inc.

Nicoll, B. and Keogh, B. (2019) 'The Unity Game Engine and the Circuits of Cultural Software', in, pp. 1–21. doi: 10.1007/978-3-030-25012-6_1.

Nintendo (1981) 'Donkey Kong'. Nintendo.

Novak, J. (2007) *Game Development Essentials: An Introduction*. 2nd edn. Delmar Learning.

Nystrom, R. (2014) *Game Programming Patterns*. Genever Benning. Available at: https://books.google.co.uk/books?id=AnvVrQEACAAJ.

O'Hagan, A. O., Coleman, G. and O'Connor, R. V (2014) 'Software development processes for games: A systematic literature review', in *European Conference on Software Process Improvement*, pp. 182–193.

Pajitnov, A. and Pokhilko, V. (1984) 'Tetris'. Nintendo.

Pan European Game Information (2003) 'PEGI'. Belgium. Available at: https://pegi.info/.

Paul, P. S., Goon, S. and Bhattacharya, A. (2012) 'History and comparative study of modern game engines', *International Journal of Advanced Computed and Mathematical Sciences*, 3(2), pp. 245–249.

Pavlovic, D. (2020) 'Video Game Genres: Everything You Need to Know', *HP*. Available at: https://www.hp.com/us-en/shop/tech-takes/video-game-genres.

Ramadan, R. and Widyani, Y. (2013) 'Game development life cycle guidelines', in *2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pp. 95–100. doi: 10.1109/ICACSIS.2013.6761558.

Rogers, S. (2014) *Level Up! The Guide to Great Video Game Design*. 2nd edn. Wiley Publishing.

Roth, E. (2020) 'What Makes a Game "Retro?"', *whatnerd*. Available at: https://whatnerd.com/what-makes-a-game-retro/.

Ruparelia, N. B. (2010) ‘Software Development Lifecycle Models’, *SIGSOFT Softw. Eng. Notes*. New York, NY, USA: Association for Computing Machinery, 35(3), pp. 8–13. doi: 10.1145/1764810.1764814.

Salen, K. and Zimmerman, E. (2003) *Rules of Play: Game Design Fundamentals*. MIT Press.

Sandeepa, P. (2021) ‘Waterfall Model vs Agile Model’, *Medium*. Available at: https://medium.com/linkit-intecs/waterfall-model-vs-agile-model-c580e389feb.

Scarratt, D. (2018) ‘The evolution of audio in videogames’, *acmi*. Available at: https://www.acmi.net.au/stories-and-ideas/evolution-audio-videogames/.

Schultz, C. P. and Bryant, R. D. (2016) *Game Testing: All in One*. Mercury Learning \& Information. Available at: https://books.google.co.uk/books?id=OD6TDgAAQBAJ.

Sherrod, A. (2007) *Ultimate 3D Game Engine Design \& Architecture*. Charles River Media (Charles River Media game development series). Available at: https://books.google.co.uk/books?id=kjy-AAAACAAJ.

Simon-Kucher & Partners (2020) *New Study: Gamers Around the World are Spending More Time and Money on Video Games during the COVID-19 Crisis, and the Trend will likely Continue Post-Pandemic*. Available at: https://www.simon-kucher.com/en-gb/about/media-center/new-study-gamers-around-world-are-spending-more-time-and-money-video-games-during-covid-19-crisis-and-trend-will-likely-continue-post-pandemic.

Sommerville, I. (2011) *Software Engineering*. Pearson (International Computer Science Series). Available at: https://books.google.co.uk/books?id=l0egcQAACAAJ.

Square Enix (2015) ‘Life Is Strange’. Square Enix, Feral Interactive.

Szulc, M. (2015) *Must Deliver Backgrounds*, *Behance*. Available at: https://www.behance.net/gallery/29946977/Must-Deliver-Backgrounds.

Taito Corporation (1978) ‘Space Invaders’. Japan: Nintendo, Atari, Taito.

Ternyak, J. and Peysakhovish, R. (2021) ‘COVID Invaders’. Available at: https://www.getonedesk.com/covid-invaders.

Toftedahl, M. (2019) ‘Which are the most commonly used Game Engines?’, *gamedeveloper.com*. Available at: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-.

UML (2015) ‘OMG Unified Modeling Language (UML)’. Available at: https://www.omg.org/spec/UML/2.5/PDF.

Unity Technologies (2005a) ‘Unity’. Denmark.

Unity Technologies (2005b) ‘Unity User Manual 2020.3 (LTS)’. Available at: https://docs.unity3d.com/Manual/index.html.

Wallach, O. (2020) ‘50 Years of Gaming History, by Revenue Stream (1970-2020)’, *Visual Capitalist*. Available at: https://www.visualcapitalist.com/50-years-gaming-history-revenue-stream/.

Ware, B. (2021) *PixelArt*. Available at: https://www.pixilart.com/.

Wijman, T. (2020) ‘The World’s 2.7 Billion Gamers Will Spend $159.3 Billion on Games in 2020; The Market Will Surpass $200 Billion by 2023’, *Newzoo*. Available at: https://newzoo.com/insights/articles/newzoo-games-market-numbers-revenues-and-audience-2020-2023/.

Wirtz, B. (2021) ‘Unity vs. Godot: Performance, Community Support, Ease of Use, and Pricing’, *gamedesigning.org*. Available at: https://www.gamedesigning.org/engines/unity-vs-godot/.

Zarrad, A. (2018) ‘Game engine solutions’, *Simulation and Gaming*. BoD--Books on Demand, pp. 75–87.

Zgeb, B. (2021) ‘Persistent data: How to save your game states and settings’, *Unity Blog*. Available at: https://blog.unity.com/technology/persistent-data-how-to-save-your-game-states-and-settings.

Appendix A - Requirements

Functional Requirements

1. System/Game Design

1.1. The system shall consist of a title scene.

1.2. The system shall consist of at least six level scenes.

1.3. Each level shall contain at least a starting view, game view and end view.

1.4. The game shall consist of a consistent story of the user trying to destroy the enemy in each level.

1.5. The system shall save the users game state at the end of each level.

1.6. The system shall ask for confirmation from the user when exiting the application.

2. Level Design

2.1. The starting view shall be the initial view when each level scene is loaded.

2.2. The starting view shall transition to the game view via user input.

2.3. The game view shall consist of an arena[16], a player, an enemy, and a Heads-Up Display (HUD).

2.4. The game view shall allow the user to pause the game given the correct input.

2.4.1. On pausing the game, the user shall be allowed to resume the game.

2.4.2. On pausing the game, the user shall be allowed to return to the title scene.

2.4.3. On pausing the game, the user shall be allowed to exit the application.

2.5. The game view shall transition to the end view when either the player or enemy dies.

2.5.1. If the player dies, a game-over text shall be displayed on the end view.

2.5.2. If the enemy dies, a level complete text shall be displayed on the end view.

2.5.3. If the enemy dies and the user is on the final level, a game complete text shall be displayed on the view.

2.6. The end view shall allow transition to a new scene.

2.6.1. If the player has died[17], the end view shall allow transition to the title scene via user input.

2.6.2. If the enemy has died[18], the end view shall allow transition to the next level scene via user input.

2.6.3. If the enemy has died and the user has completed the final level, the end view shall allow transition to the title scene.

[16] Arena is defined as the area in which the player and enemy are contained within. If any bullets or projectiles hit the arena walls they are destroyed.

[17] Player death is denoted by the player's health value being equal to 0.

[18] Enemy death is denoted by the enemy's health value being equal to 0.

3. Title Scene

3.1. The title scene shall consist of at least a main menu view that allows the user to navigate through different options.

3.1.1. The main menu view shall be the initial view of the title scene.

3.2. The main menu view shall provide the option for the user to start a new game.

3.2.1. If a game can be loaded, the system shall ask the user for confirmation on starting a new game.

3.2.2.1 If the user denies starting a new game, the system shall return to the main menu view of the title scene.

3.3. The main menu view shall provide the option for the user to load a new game.

3.3.1. If no game can be loaded, the system shall display an error message and allow the user to return to the main view of the title scene.

3.4. The main menu view shall allow the user to view the input controls of the game.

3.4.1. If the user chooses to view the controls, the system shall display the keys for player movement, player shooting and to pause the game.

3.5. The main menu view shall allow the user to view the highest scores achieved.

3.5.1. If the user selects to view the high scores, the system shall display the top ten highest scores achieved.

3.6. The main menu view shall provide a way for the user to exit the application.

3.6.1. If the user selects to exit the application, the program must shut down.

3.7. The main menu view shall provide a way for the user to adjust sound and resolution settings.

3.7.1. The system shall allow the user to adjust the game's volume.

3.7.2. The system shall allow the user to change the resolution of the application.

3.7.3. The system shall allow the user to toggle on or off the display being full screen.

4. Loading Scenes

4.1. The system shall display a loading screen between scene transitions.

4.2. The loading screen must display to the user that the new scene is loading.

4.3. The loading screen shall display its loading progression to the user.

4.4. The loading screen must be displayed until the new scene has fully loaded.

4.4.1. When the new scene has fully loaded, the active view shall switch from the loading screen to the initial view of the new scene.

5. Player

5.1. The player must be able to move left and right given the appropriate user input.

5.2. The player must be able to shoot bullets from the chosen gun given the appropriate user input.

5.3. The user shall be given options to upgrade the player on the completion of each level.

5.3.1. The user could upgrade the player's weapon.

5.3.2. The user could increase the player's movement speed.

5.3.3. The user could increase the player's maximum health.

5.3.4. The user could gain shield for the player.

5.3.4.1. If the player has already gained shield, the user could top-up the shield to its maximum value.

5.3.4.2. If the player has shield, the player shall display the presence of an intact shield via a bar and/or animation.

5.4. Only one player shall be contained on the game view of each level.

5.5. The player shall display movement animations when the user chooses to move the player.

5.6. The player shall display a death animation when its health drops to 0.

6. Enemy

6.1. Only one enemy shall be contained on the game view of each level.

6.2. The enemy's behaviour must differ in each level.

6.3. The enemy's difficulty shall increase as the user progresses through the levels.

6.4. The enemy shall move within the arena boundaries.

6.5. The enemy shall shoot projectiles.

6.6. The enemy could implement elements of randomness in both its movements and shooting patterns.

6.7. The enemy could implement different modes where its behaviour differs from its default characteristics.

6.7.1. The enemy shall display different animations when switching modes.

6.8. The enemy's health could increase as the user progresses through the levels.

6.9. The enemy shall display a death animation when its health drops to 0.

7. Health System

7.1. The player's base health shall start at 100.

7.2. The player's health is refilled on the completion of each level.

7.3. The player must take damage when hit by an enemy projectile.

7.3.1. If the player has an intact shield[19], the shield level must decrease.

7.3.2. If the player's shield is not intact, or the player does not possess a shield, the player's health must decrease.

7.3.3. The player must die when its health drops to 0.

7.3.4. The player could take increases amount of damage as the user progresses in levels.

7.4. The enemy must lose health when hit by a player bullet.

7.4.1. The enemy must die when its health drops to 0.

[19] An intact shield has a shield value greater than 0.

7.4.2. The amount of decrease in enemy health must depend on the player's current weapon.

8. Heads-Up Display (HUD)

8.1. The game view must contain a HUD for displaying game information to the user

8.1.1. The HUD must display information concerning the player's health, the player's shield (if the player has shield), the enemy's health, and the score.

8.2. The HUD shall contain a health-bar displaying the current state of the player's health.

8.3. If the player has shield, the HUD shall contain a shield-bar displaying the current state of the player's shield.

8.4. The HUD shall contain a health-bar displaying the current state of the enemy's health.

8.5. The HUD shall display the current score of the user.

9. Scoring System

9.1. The user's score shall increase if the player's bullet hits the enemy.

9.2. The user's score shall decrease if the player's bullet does not hit the enemy.

9.3. The user's score shall decrease if the enemy's projectile hits the player.

10. Game Mechanics/Rules

10.1. The player and enemy in each level must only move within the confines of the arena.

10.2. Bullets and projectiles must be destroyed when colliding with either the player, enemy or arena walls.

10.3. The player bullets must not collide with the enemy projectiles.

10.4. The enemy projectiles must not collide with the player bullets.

11. Saving

11.1. The user's game state shall be saved at the end of each level.

11.1.1. The game state shall consist of at least the player's name, user's score, current weapon type, current level number, player's health, player's state of shield, and current movement speed.

11.2. The game state shall be saved in a file and written in JSON format.

11.3. The game state file shall be created on completion of the first level.

11.4. The game state file shall be read if the user chooses to return to the saved game state.

11.5. The game state file shall be deleted on completion of all the levels and when the player dies.

11.6. The user's score shall be saved at the end of the game[20] only if the score is within the top ten of the highest scores.

12. Audio

12.1. The game audio shall contain the player and enemy deaths.

[20] End of the game defined by two conditions: the player dying or the user completing all the levels.

12.2. The game audio shall contain the player shooting.

12.2.1. Each weapon shall have a different sound.

12.3. The game audio shall contain of background music.

12.3.1. The background music could change depending on the current state of the game (i.e., different music when battling the enemy).

13. High Score

13.1. The highest scores file shall be created on the first instance of the user opening the application and written in JSON format.

13.1.1. The highest scores file shall be filled in with default name and score values when first created.

13.1.2. The highest score file shall contain a maximum of ten high score entries[21].

13.1.2.1. A high score entry shall contain at least the user's name and score at the end of the game.

13.2. The high scores display shall read from the highest score file and display its content.

13.3. The high scores display shall contain the rank, the user's name and the score achieved.

13.4. The high scores display shall be displayed in descending order (highest score first).

Non-Functional Requirements

1. Performance

1.1. The system shall load the application within 5 seconds.

1.2. The system shall load each scene within 3 seconds.

1.3. The system shall load each view in a scene within 1 second.

1.4. The system shall load a saved game within 3 seconds.

1.5. The system shall write to necessary files within 2 seconds.

1.6. The system shall respond to user inputs within 0.3 seconds.

1.7. The system's minimum frame rate shall be 20 frames per second.

1.8. The system's average frame rate shall be a minimum of 30 frames per second.

2. Space

2.1. The system shall not exceed 500MB of memory.

2.2. The system shall be able to run with a minimum of 1024MB of RAM.

2.3. Files the system creates shall not exceed 1MB of memory.

3. Portability

3.1. The system shall be compatible with Windows.

[21] A high score entry defined as one instance of the user's game. (i.e. If the player dies or has completed all the levels, the user's name and score considered an entry to the high score list).

3.2. The system shall be compatible with MacOS.

3.3. The system shall be compatible with Linux.

3.4. The system shall ensure its usability with newer versions of the above stated operating systems.

4. Reliability

4.1. The system shall operate smoothly without crashing.

4.2. The system shall maintain a mean time between failures (MTBF) of 100 hours.

5. Usability

5.1. The system shall be suitable for people over the age of 5.

5.2. The system shall be easy to use. The users should understand the system controls within 2 minutes.

5.3. The system UI shall be clean and compatible with desktop screens.

5.4. The system UI shall have a consistent layout and colour scheme.

5.5. The system should use the same language (English) throughout.

5.6. The maximum system audio shall not exceed user audio threshold.

6. Maintainability

6.1. The system code shall be well documented.

6.2. The code written for the system must be maintainable with good code metrics.

6.2.1. The system must maintain a maximum of 3 on the depth of inheritance tree (DIT).

6.2.2. The system code must maintain low coupling.

7. Scalability

7.1. The system shall be developed such that more levels can be added with ease.

7.2. The system shall be developed such that more weapons can be added with ease.

7.3. The system shall be developed such that additional enemies can be added with ease.

8. Implementation

8.1. The system shall be developed using the Unity Engine.

8.2. The system code shall be developed in C#.

8.3. The system application shall require no other dependencies other than what has been encompassed within the system.

Appendix B – Gantt Chart

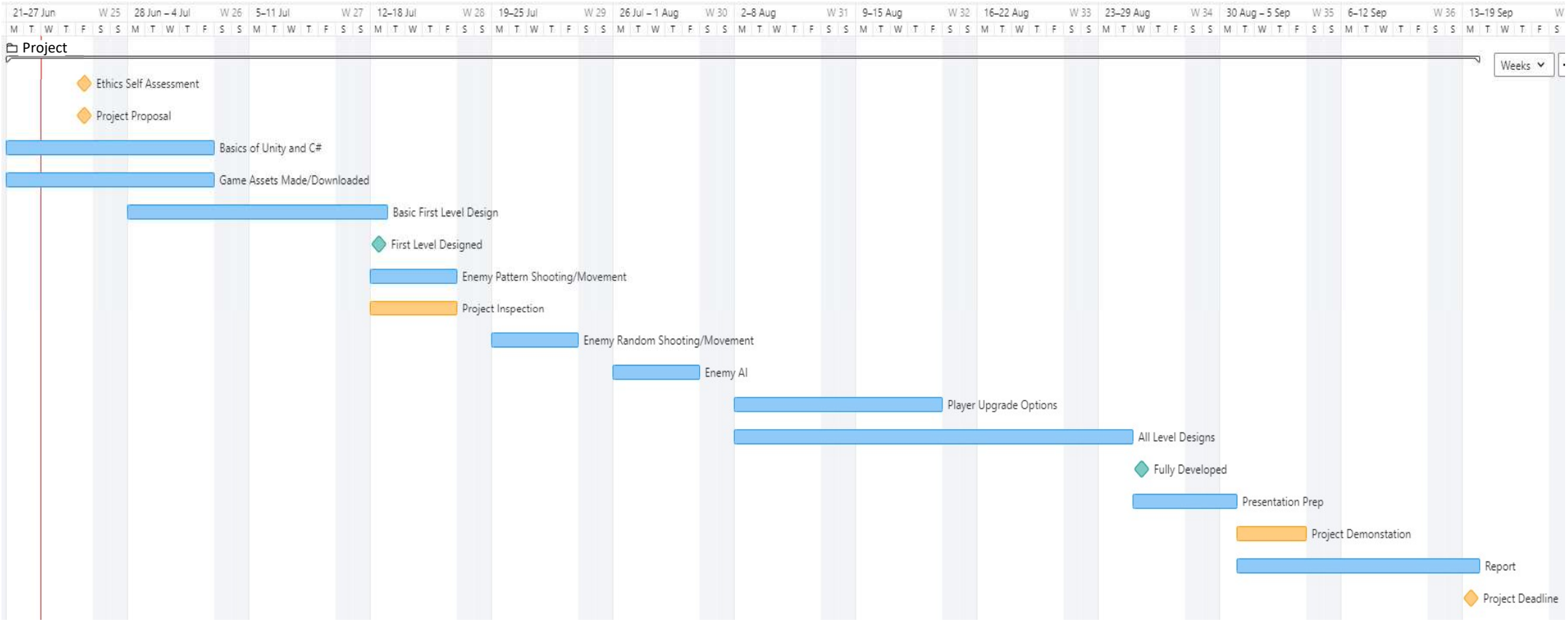
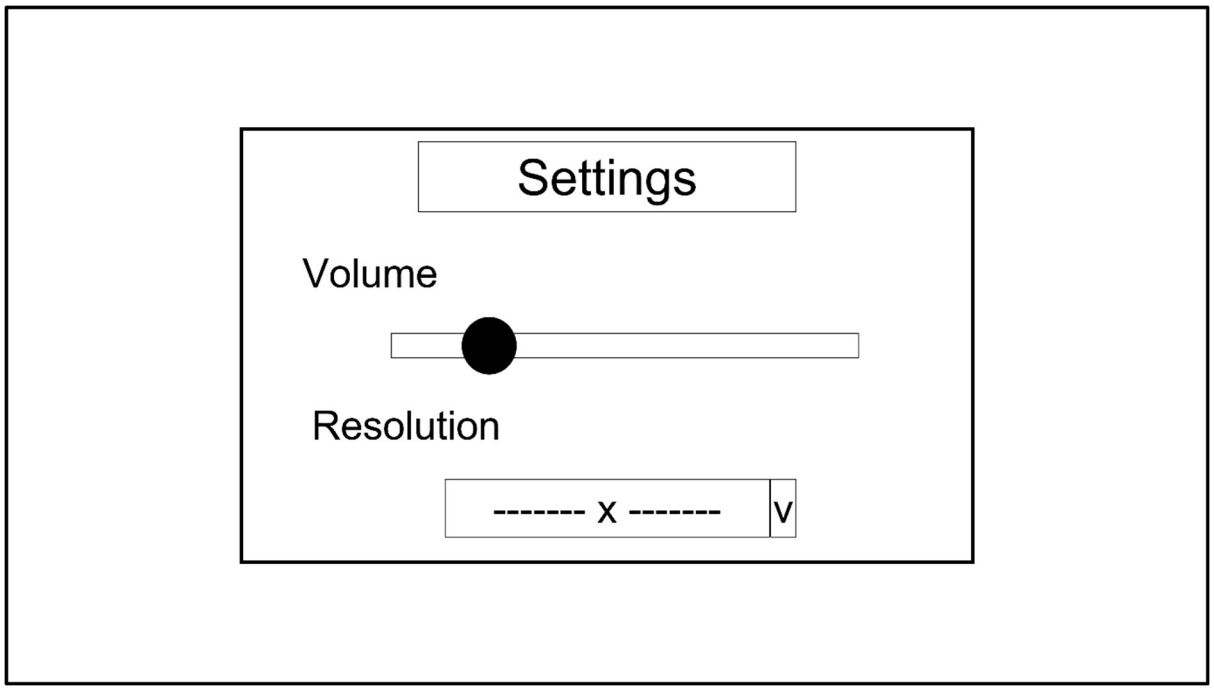
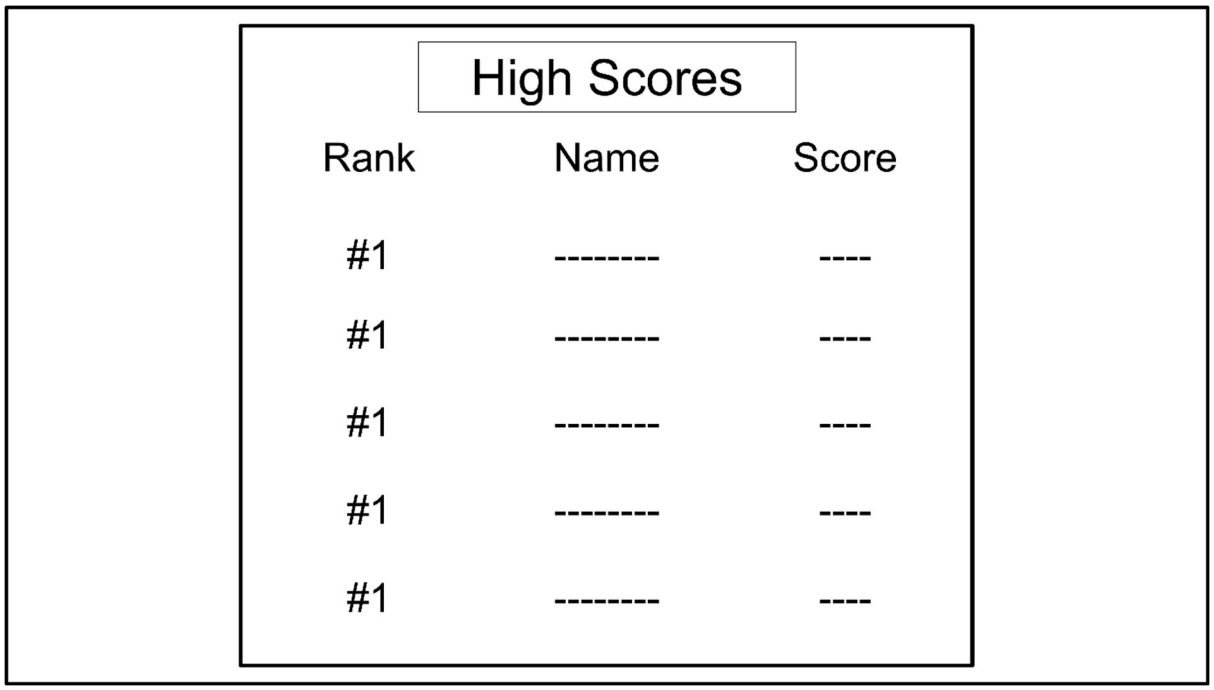


Figure 26: Gantt Chart of the project

Appendix C – Wireframes



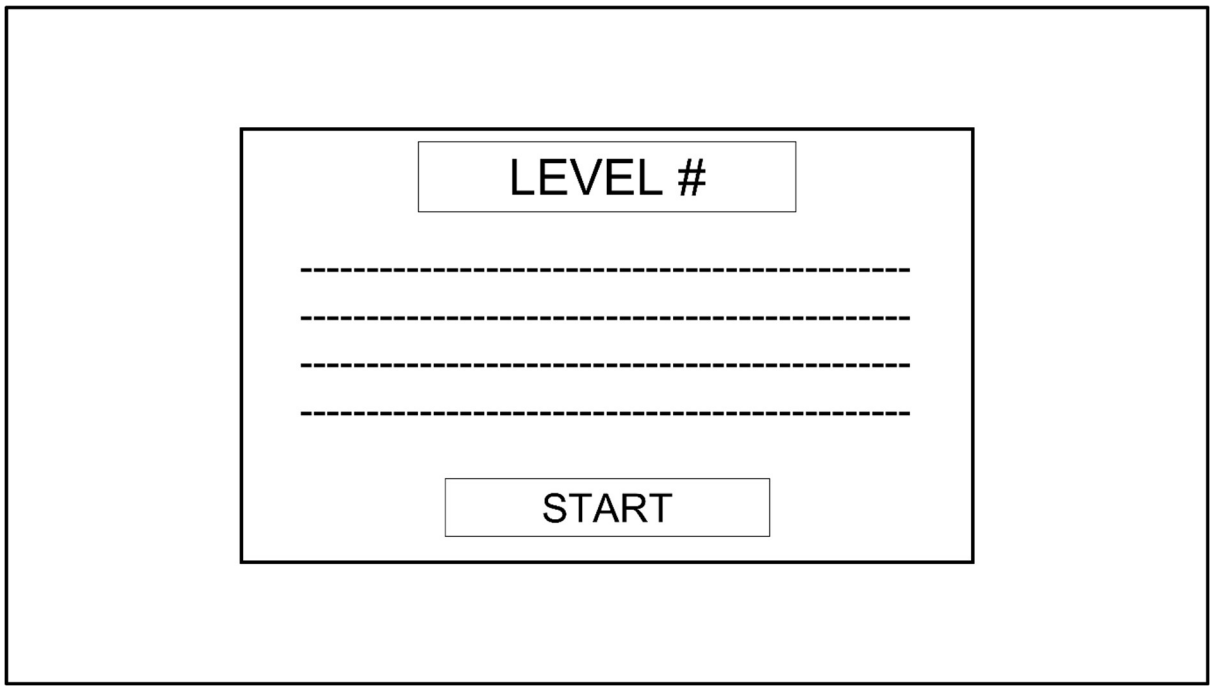

High Scores
Rank
Name
Score
#1

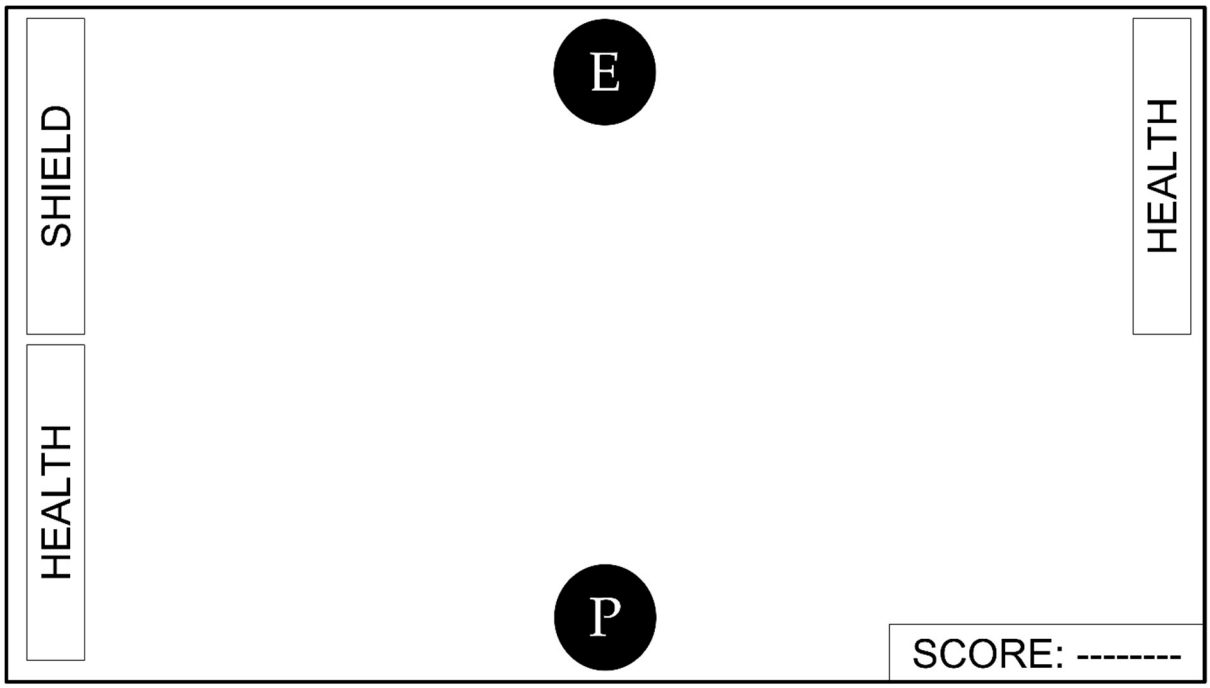
#1

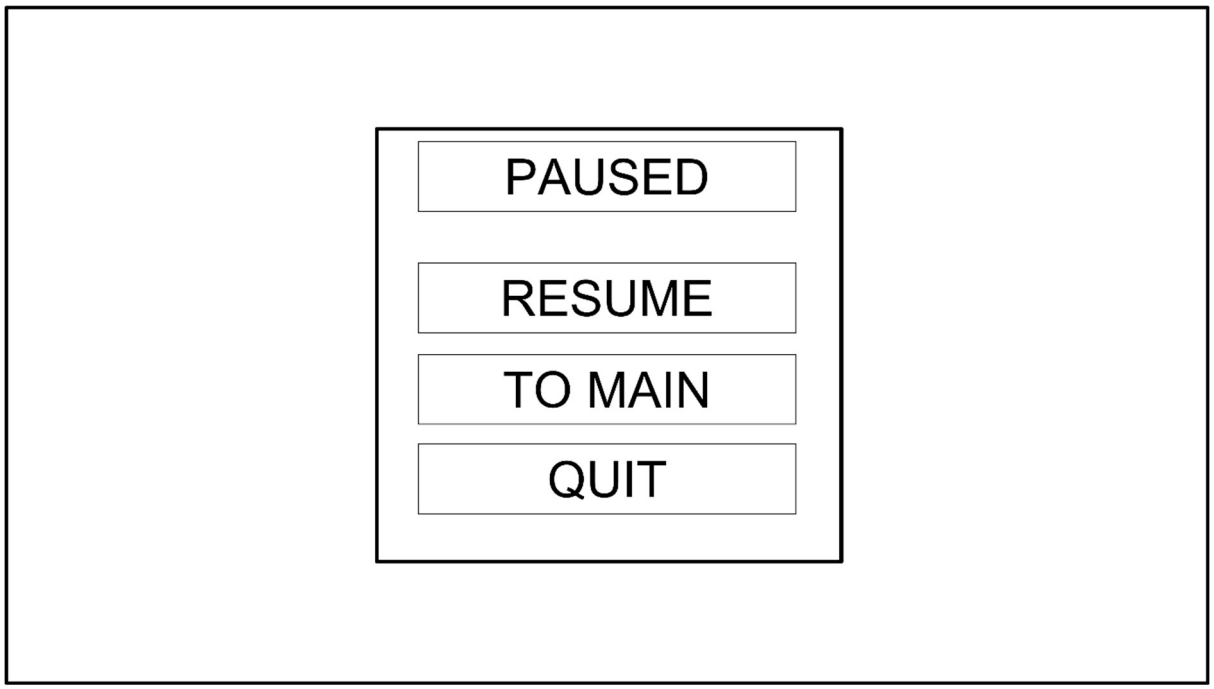
#1

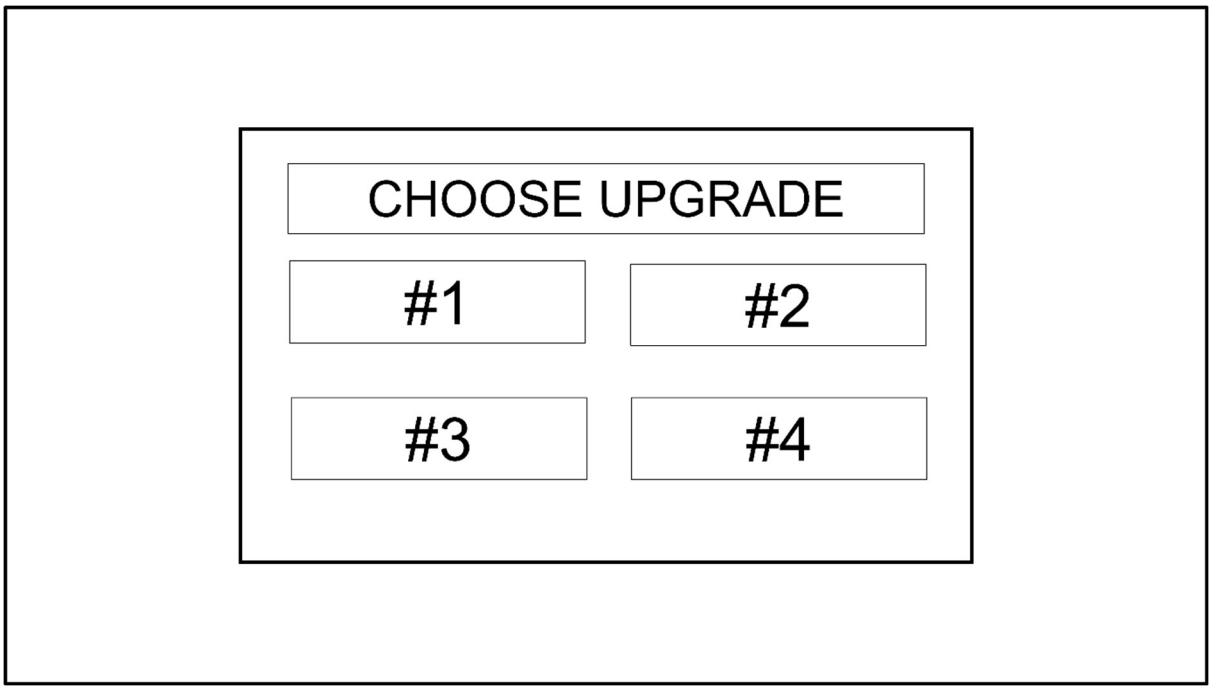
#1

#1


LEVEL #
START


SHIELD
E
HEALTH
HEALTH
P
SCORE: --------


PAUSED
RESUME
TO MAIN
QUIT


CHOOSE UPGRADE
#1
#2
#3
#4

Appendix D – System Screenshots and Art

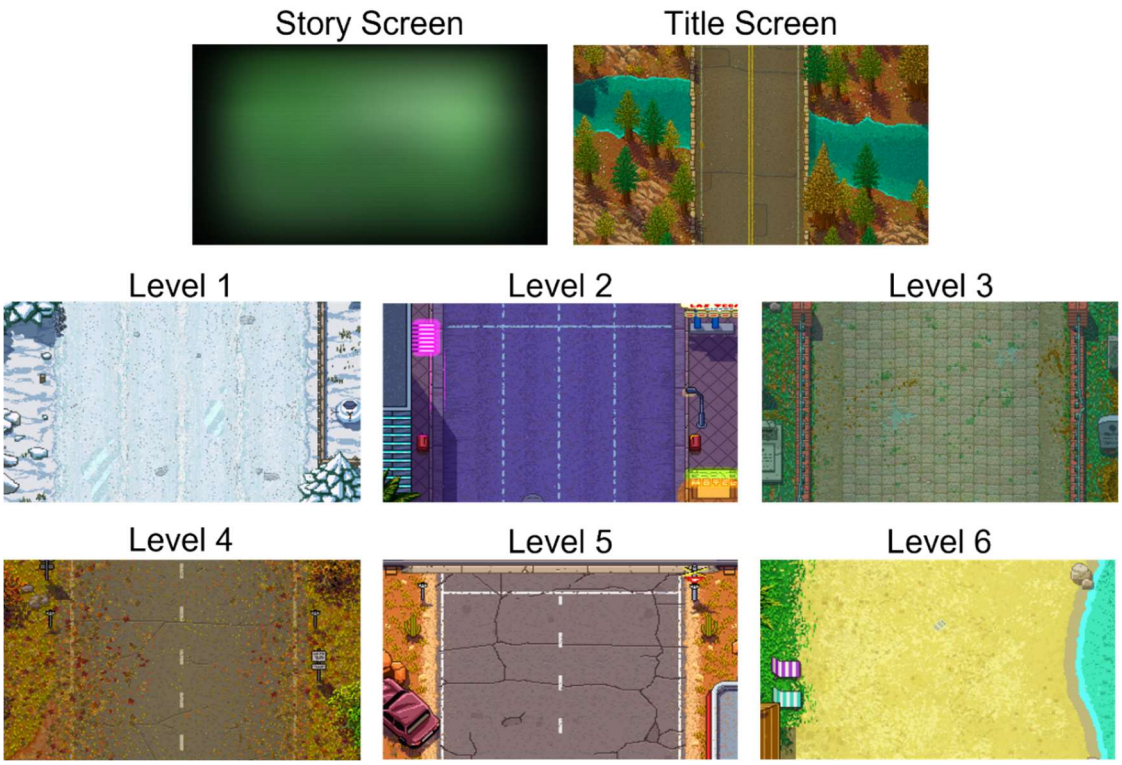


Figure 27: Background artwork in every scene adapted from (GorillazXD, 2016) *for the story screen and* (Szulc, 2015) *for the other backgrounds.*

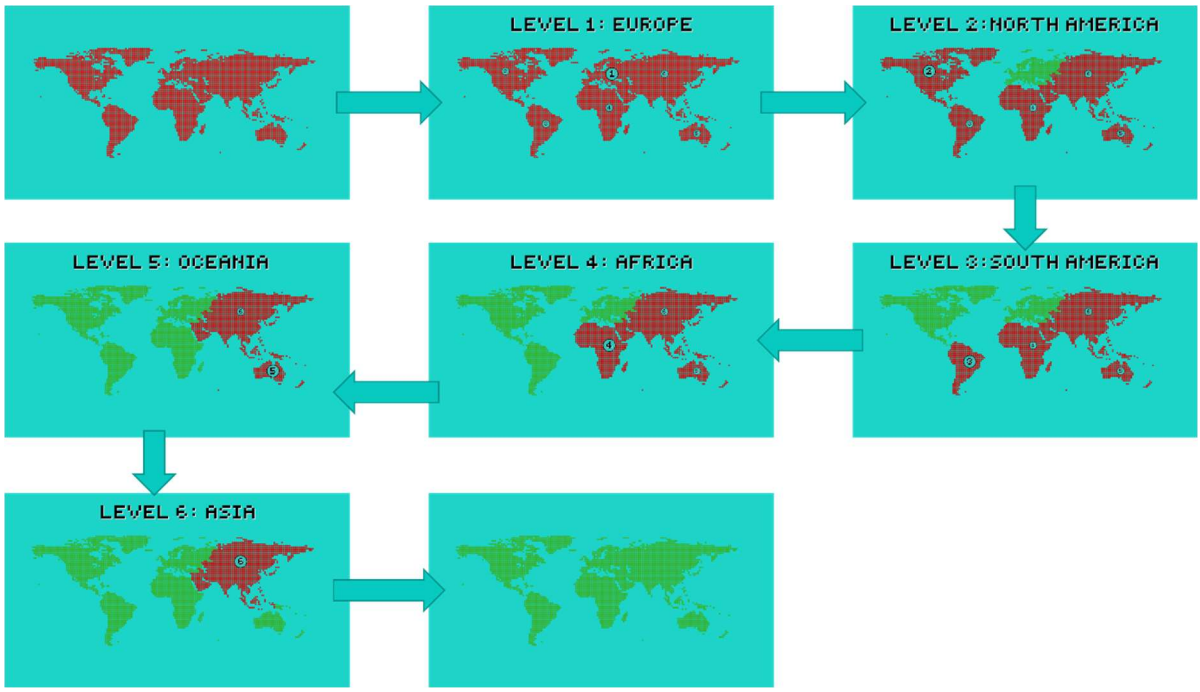


Figure 28: The different loading screens as the user advances through the levels[22]

[22] Adapted from https://pixelmap.amcharts.com/

Figure 29: All Title Scene views

Figure 30: All Level Scene views for Level 4 and the congratulations end screen for when the game is completed on Level 6

Appendix E – Testing

Edit and Play Mode Tests

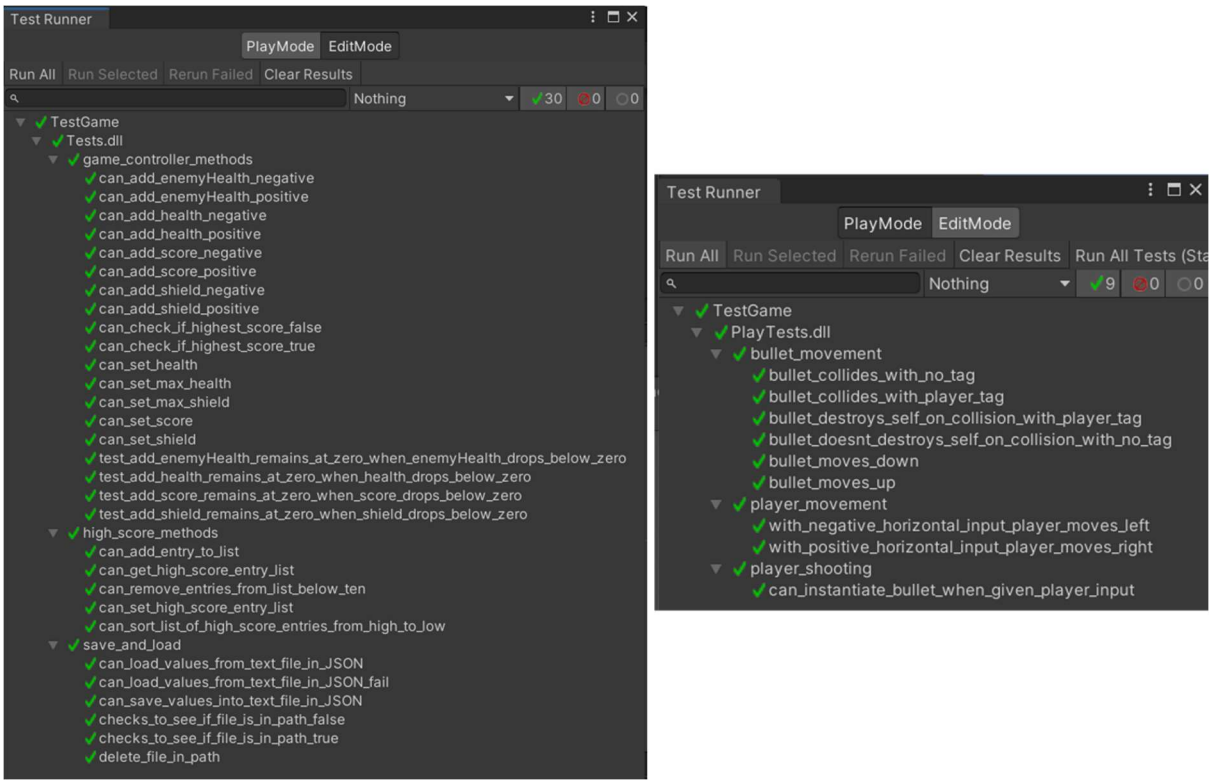


Figure 31: Edit mode (left) and Play mode (right) testing of some functionalities of the game

Test Cases

ID	Test Case	Expected Outcome	Pass/Fail
Program Launch and Story Scene			
1	Program is launched	Program opens and story screen of text is displayed	Pass
2	User clicks any other key other than enter	Game stays on story screen	Pass
3	User clicks enter key	Game shows loading screen before title scene	Pass
Title Scene			
4	Loading the title scene	Title screen displayed with background, game title, exit button, settings button, new game button, load game button, how to play button and high scores button	Pass
5	Title scene loaded	User hears the background music	Pass

6	User clicks ‘quit’ button	Quit view confirming quitting the game is shown	Pass
7	User clicks ‘no’ on quit view	Game return to main title screen	Pass
8	User clicks ‘yes’ on quit view	Application exits	Pass
9	User clicks ‘settings’ button	Game show’s settings view with sliders set to 75	Pass
10	User drags background slider to maximum	Value besides slider shows 100 and background music get louder	Pass
11	User drags SFX slider to maximum	Value besides slider shows 100	Pass
12	User drags background slider to minimum	Value besides slider shows 0 and background music is mutes	Pass
13	User drags SFX slider to maximum	Value besides slider shows 0	Pass
14	User toggles fullscreen to disable	The application is now in windowed	Pass
15	User toggle resolution to 1080	Application window is of size 1920x1080	Pass
16	User toggle resolution to 720	Application window is of size 1280x720	Pass
17	User toggle resolution to 540	Application window is of size 960x540	Pass
18	User toggle resolution to 480	Application window is of size 852x480	Pass
19	User toggle full screen to enable	The application is now full screen	Pass
20	User clicks ‘back’ on settings view	Game returns to main title screen	Pass
21	User clicks ‘high scores’ button	Game shows high scores view with default player names and values	Pass
22	User clicks ‘back’ on high scores view	Game returns to title screen	Pass
23	User clicks ‘how to play’ button	Game shows how to play screen	Pass
24	User clicks ‘back’ on how to play view	Game returns to title screen	Pass
25	User clicks ‘load game’ button with no game to load	Load error view shows up	Pass
26	User clicks ‘ok’ on load error view	Game returns to title screen	Pass
27	User clicks ‘load game’ button with a game to load	Game transitions to new level scene	Pass

28	User clicks on screen anywhere other than where there is a button	Game does nothing	Pass
29	User clicks 'new game' button	Game shows enter name view	Pass
30	User presses 'enter' with empty name field	Game transitions to level one and name says 'Commander'	Pass
31	User clicks 'enter' button with empty name field	Game transitions to level one and name says 'Commander'	Pass
32	User presses 'enter' with a non-empty name field	Game transitions to level one and shows the name the user typed	Pass
33	User clicks 'enter' button with a non-empty name field	Game transitions to level one and shows the name the user typed	Pass
34	User types more than 15 characters	Game limits user to only 15 characters within the field	Pass
Level Scenes (generic to all level scenes)			
35	On scene load	Game shows background and starting view	Pass
36	User clicks 'start' button on start view	Game shows game view	Pass
37	User presses 'enter' on start view	Game shows game view	Pass
38	On showing the game view	Game view has player, enemy, player and enemy health bars, score, pause button and shield bar if upgrade was chosen. Background music changes.	Pass
39	No action	Player does not move or shoot. Enemy is active. Enemy shooting sound is heard each time it shoots	Pass
40	User presses 'A' on game view	Player moves to the left and the animation plays	Pass
41	User presses 'D' on game view	Player moves to the right and the animation plays	Pass
42	User presses left arrow on game view	Player moves to the left and the animation plays	Pass
43	User presses right arrow on game view	Player moves to the right and the animation plays	Pass
44	User presses space on game view	Game spawns a bullet from the player's weapon and bullet travels straight up the screen. Sound played of player shooting	Pass

45	User presses left mouse click on game view	Game spawns a bullet from the player's weapon and bullet travels straight up the screen. Sound played of player shooting	Pass
46	User clicks 'pause' button on game view	Game shows pause view on top of game view. All actions on the game view are paused	Pass
47	User presses 'esc' on game view	Game shows pause view on top of game view. All actions on the game view are paused	Pass
48	User clicks 'resume' on pause view	Game returns to game view in the same state before pausing	Pass
49	User clicks 'return to main menu' on pause view	Game shows return to main confirmation view	Pass
50	User clicks 'yes' on return to main confirmation view	Game transitions to title scene	Pass
51	User clicks 'no' on return to main confirmation view	Game returns to pause view	Pass
52	User clicks 'quit' on pause view	Game shows quit confirmation view	Pass
53	User clicks 'yes' on quit confirmation view	Application closes	Pass
54	User clicks 'no' on quit confirmation view	Game returns to pause view	Pass
55	Player gets hit by enemy projectile	Player health-bar or shield bar decreases. Value next to the bar also decreases. Score decreases by 3	Pass
56	Player hits enemy with bullet	Enemy health-bar and value next to it decreases. Score increases by 20	Pass
57	Player misses enemy with bullet	Score decreases by 1	Pass
58	Player collides with left wall	Player can no longer move left	Pass
59	Player collides with right wall	Player can no longer move right	Pass
60	Player has shield	Shield bar shown next to player health bar with shield value above. Player has pink outline around it	Pass
61	Player dies by health going to zero.	Player death animation and sound is played. Level shows game over view after animation and sound completes	Pass
62	User clicks 'return to main menu' on game over view	Game transitions to title scene	Pass

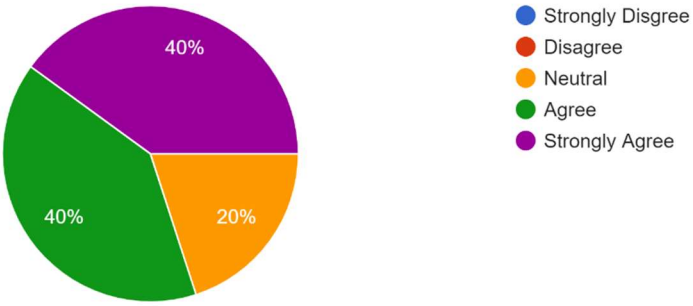
63	User clicks 'quit' on game over view	Game shows quit confirmation view	Pass
64	Enemy dies by health going to zero	Enemy death animation and sound is played. Level shows end view (or game complete view if final level) after animation and sound	Pass
65	User clicks any upgrade on end view	Game transitions to next level scene	Pass
Upgrades			
66	User chooses weapon upgrade and currently has pistol	On following level, player holds dual pistols and can shoot two bullets	Pass
67	User chooses weapon upgrade and currently has dual pistol	On following level, player holds rifle and can shoot faster	Pass
68	User chooses weapon upgrade and currently has rifle	On following level, player holds shotgun and can shoot three bullets	Pass
69	User chooses speed boost	On following level, player moves faster	Pass
70	User chooses health upgrade	On following level, health increases by 40	Pass
71	User chooses shield upgrade and currently has no shield	On following level, shield bar shown on HUD and pink outline around player	Pass
72	User chooses shield upgrade and has shield	On following level, shield bar is full	Pass
73	User chooses no upgrade	On following level, player has identical characteristics as the previous level	Pass
Loading Screens			
74	Transitioning from one scene to another	Loading screen shows up with Loading text and percentage of how much has loaded	Pass
75	Transitioning from story scene to title scene	Loading screen shows map all red with no headings	Pass
76	Transitioning from any level's scene back to the title scene	Loading screen shows map all green with no headings	Pass
77	Transitioning from title scene to level one	Loading screen shows level one heading on loading screen	Pass
78	Transitioning from level one to level two	Loading screen shows level two heading on loading screen	Pass

79	Transitioning from level two to level three	Loading screen shows level three heading on loading screen	Pass
80	Transitioning from level three to level four	Loading screen shows level four heading on loading screen	Pass
81	Transitioning from level four to level five	Loading screen shows level five heading on loading screen	Pass
82	Transitioning from level five to level six	Loading screen shows level six heading on loading screen	Pass

Appendix F – Evaluation

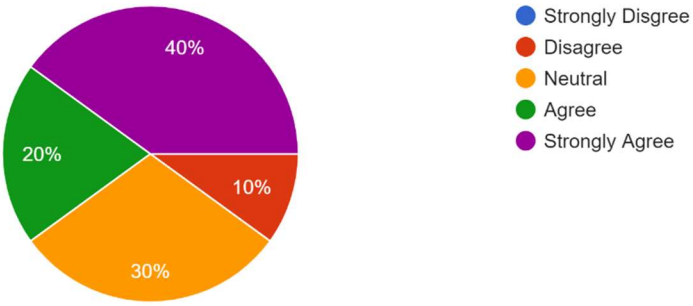
COVID Destroyer is an enjoyable game

10 responses



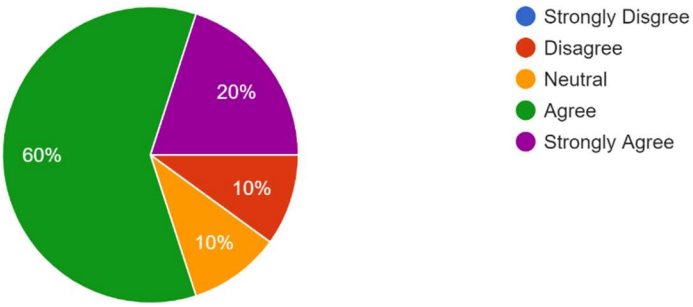
COVID Destroyer has a good game story

10 responses



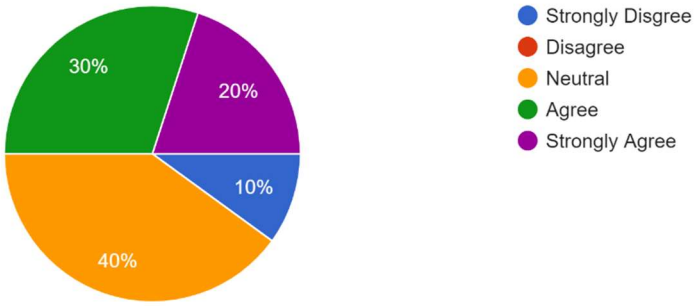
COVID Destroyer has a good level of overall difficulty

10 responses



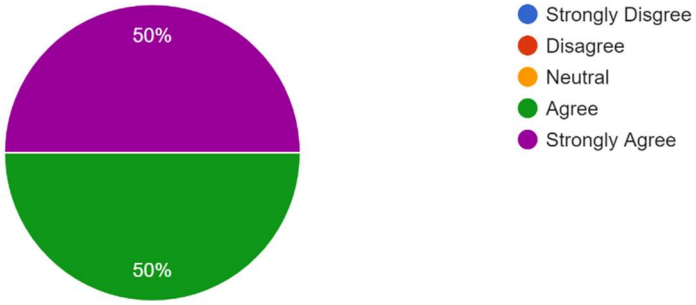
Each level of COVID Destroyer increases the difficulty by a good amount

10 responses



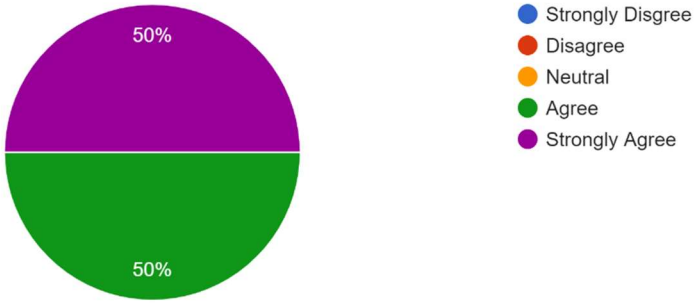
It is easy to navigate COVID Destroyer

10 responses



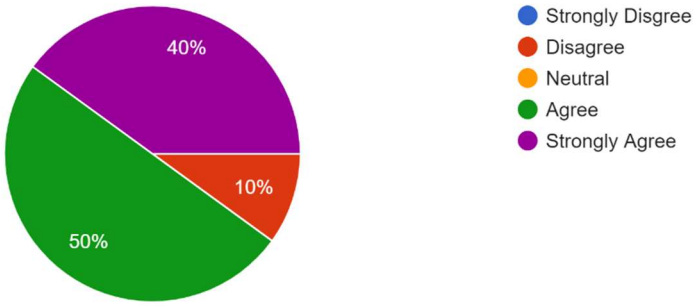
The player controls in COVID Destroyer are intuitive and easy to use

10 responses



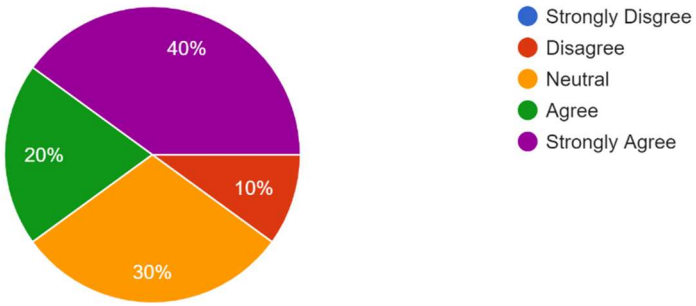
COVID Destroyer ran smoothly, without any lag or glitches

10 responses



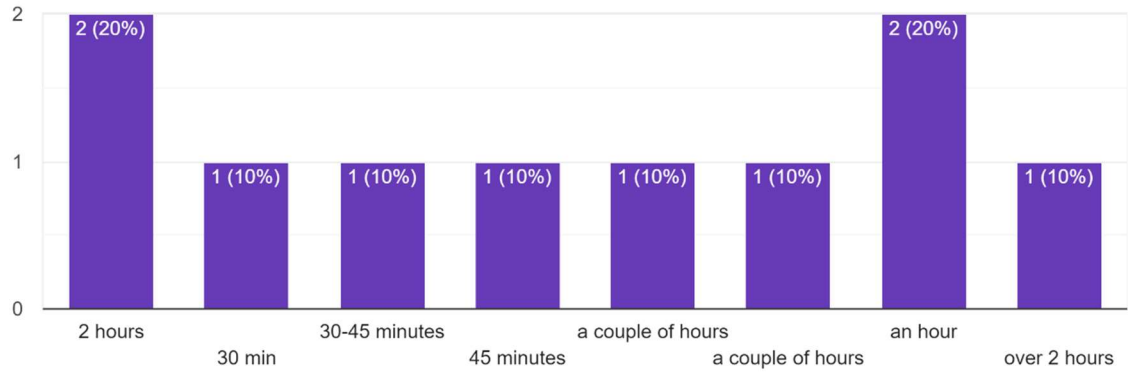
I would play covid destroyer again

10 responses



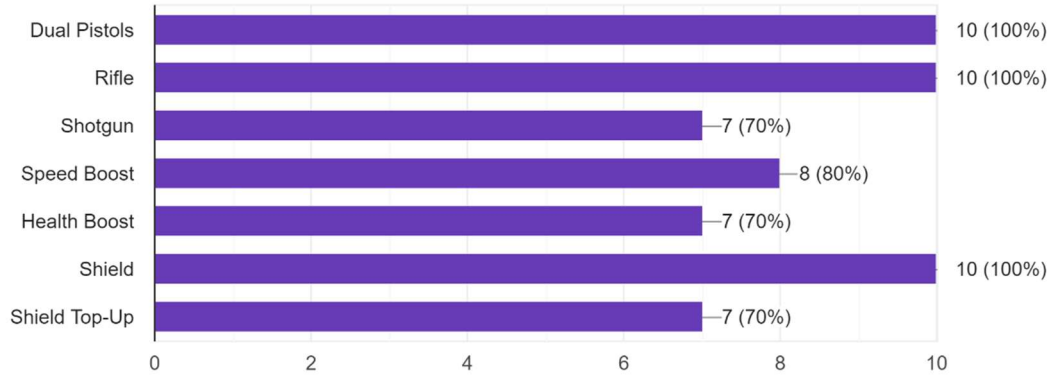
How long did you play COVID Destroyer for

10 responses



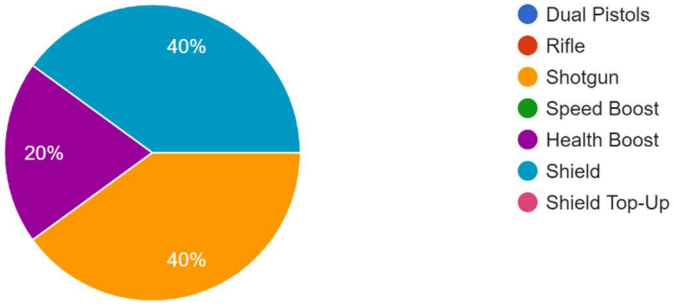
What player upgrades did you use

10 responses



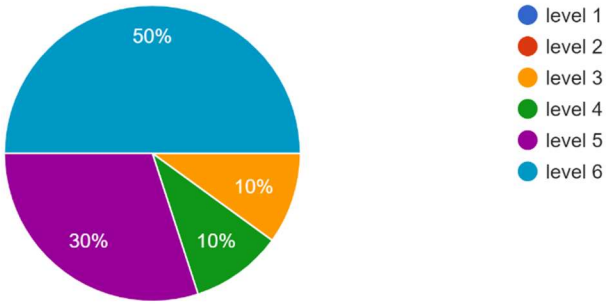
What player upgrade was your favourite

10 responses



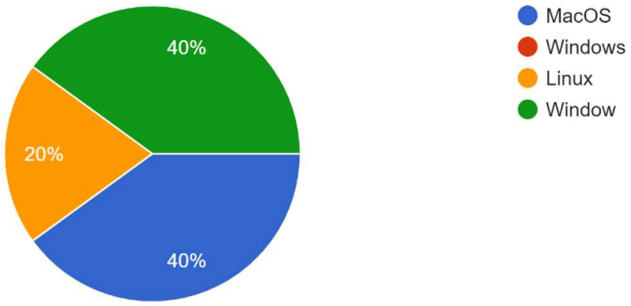
Which level were you able to advance to

10 responses



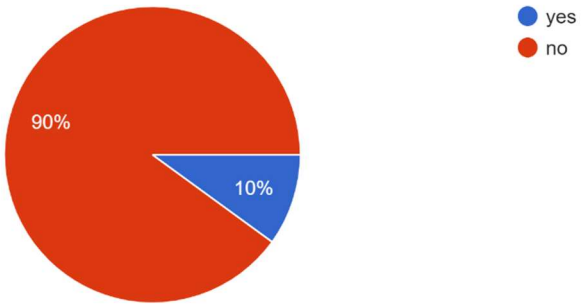
What platform did you run covid destroyer on

10 responses



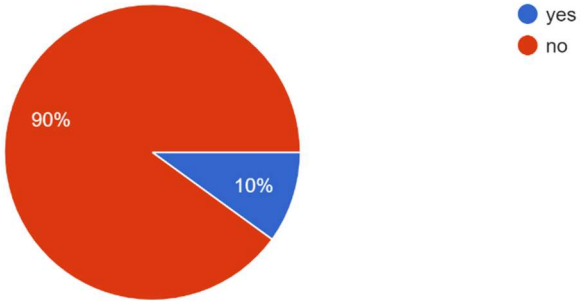
Did COVID Destroyer crash at any point during use

10 responses



Did you encounter any bugs or glitches while using COVID Destroyer

10 responses



if yes, please describe them below

1 response

when trying to return to the main menu after the player dying the game would glitch out and crash

Are there any improvements you would like to see in COVID Destroyer

8 responses

more upgrade options would be cool

The speed boost doesnt really do anything, it should be faster

better upgrades needed to beat the virus. not enough upgrades for for level 4 to get to 5

no i thought it was great!

the bugs need to be fixed and it would be a really good game

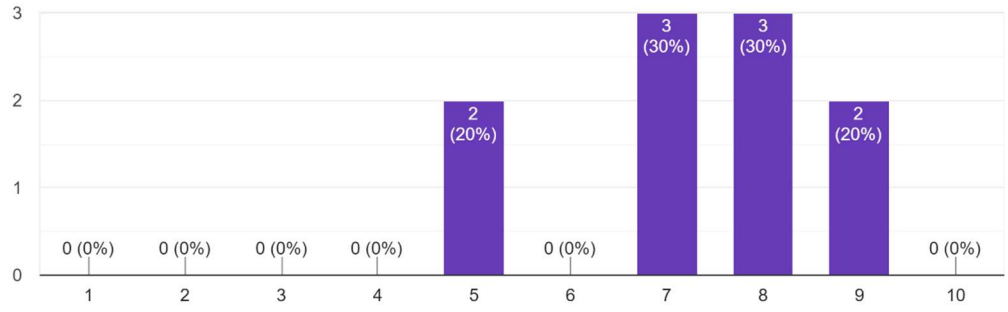
The health bars should display a numerical value for health to better show users the state of both the enemy and the player

There should be some kind of indication on the screen that the player has been hit

more levels

How would you rate COVID Destroyer overall out of 10

10 responses



Additional Comments?

7 responses

i really liked the game. If there was a part two that came out with different places and upgrades i would definetly play

the jump in difficulty between level 3 and 4 too high. if this was better i would have liked the game more

The different enemy modes were really good and added variety to the game, making it a nice challenge. i especially liked the fire when it turned red

would rate it higher but it still has a bug

good game, still cant beat the final level though!!!

The game was okay. I dont like that it is based on the pandemic

would be better if it wasnt so short. More levels would make the game better

Appendix G – GitLab Repository

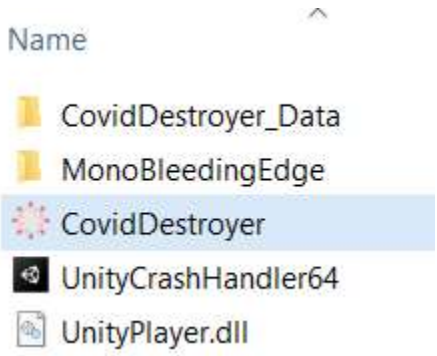
Address: https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2020/mxs486

Files in repository:

- CovidDestroyerWindow: Windows build of the game in zip file
- CovidDestroyerMac: macOS build of the game in zip file
- CovidDestroyerLinux: Linux build of the game in zip file
- TestGame: the unity project for the edit and play mode testing
- GameProject: Full Unity project before builds containing the code, assets and all that was used to build the game

How to run the game on Desktop:

1. Click the link to the repository
2. Download the zip file that corresponds to your platform (Windows, macOS, Linux)
3. Extract the zip file. On Windows the content inside the folder should be:



4. Run the CovidDestroyer.exe file
5. Enjoy the game

Note: Permission may need to be granted to the file for it to run