Making a Cross-platform Local Multiplayer Game

CS39440 Major Project Report

Author: Greg Card (glc3@aber.ac.uk)
Supervisor: Dr. Neal Snooke (nns@aber.ac.uk)

6th May 2022

Version 1.0 (Final)

This report is submitted as partial fulfilment of a BSc degree in
Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name: Greg Card

Date: 06/05/2022

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Greg Card

Date: 06/05/2022

Acknowledgements

I am grateful to my supervisor Neal Snooke for his advice and support through the projects development and writing of this report. I am also grateful to David Hunter who provided helpful feedback after the mid-project demonstration.

I'd like to thank my partner Kaitlyn James who has continually kept me motivated throughout while ensuring I was in a clear mindset during the more challenging moments. I would also like to thank Matthew Hibbin who helped test the multiplayer features and provide insightful feedback which has allowed the game to reach the standards it has.

Abstract

This project features a cross-platform local multiplayer 2D game featuring a blend of features taken from classic twin stick shooter and tower defence genre of games. The game is written in JavaScript and does not include the use of a game engine or other third-party libraries other than standard APIs offered in modern internet browsers such as Firefox or Chromium. Inputs are taken from a modern game controller design, in this case an Xbox One controller.

The aim of the game is to achieve a high score, by eliminating waves of enemy tanks before the player's base inevitably gets destroyed by enemy advancements. A key feature is the inclusion of local multiplayer which allows up to two players to simultaneously work together to achieve a high score as a team effort.

In order to facilitate their defence needs, destroyed enemies often drop coins which upon collection allows the players to purchase walls which can be used to force the enemies to take a multitude of different routes towards the base. Enemies employ a robust A-Star path finding algorithm so that they always find a potential route no matter how convoluted the players may make it. The enemies come in two varieties; one which simply moves towards the base and explodes upon contact and another, which can also shoot back at the player while moving towards their base.

The game can run on many platforms which contain a modern web browser and a method of connecting a game controller which has been accomplished by employing web technologies and code which handles platform differences.

Table of Contents

1. Background, Analysis and Process

1.1 Background

In December 2021, I decided that I would make a game without a game engine for my final year project. One major question many people may instantly ask is "Why would you want to make a game without using a game engine?" This is a good question because it does pose a good point as to whether there are any advantages to making a game without using a game engine such as Unity (Unity Technologies, 2022) and Unreal (Epic Games, Inc, 2004-2022). This question is discussed further in section 1.1.2.

The game I will be making is a top-down twin stick shooter with tower defence mechanics which will discussed in greater detail in the following section.

1.1.1 Choosing a Game to Make

While it was known by January that a game would be my major project, there was still no final decision as to what kind of game would be designed. All that was known is that it would be 2D and that it would take inspiration from "Wii Play – Tanks!" a relatively simple game which released on Nintendo's Wii console on December 8th, 2006. Wii Play Tanks was chosen as a source of inspiration due to it being a favourite of mine while also having relatively simple mechanics.

One key design goal was to include local multiplayer which became a popular feature over lockdown despite people being further apart. This was possibly due to the introduction of new streaming technology such as Steam Link (Valve Corporation, 2015) which allows people to connect from afar which happened to coincide with more people staying at home. Local multiplayer typically means that two input devices would be connected to the computer, while both players look at the same screen. Each player would be controlling their own character, or whatever controllable feature would be in the game using their own input device, such as controller.

Potential Ideas Included:

- A frogger type game, however players would start in the middle and work their way out in four different directions. However, this idea was not too original, and it would have been hard to integrate local multiplayer.
- A scrolling space shooter game where two players could face off incoming enemies which would fire missiles back at the players. Each player could use powerups to help fend off enemy attacks or to increase their own attack power. However, this idea was not used because it was deemed too simple and again, not original.

In the end, it was decided that the game would be a blend between my own personal favourite genre of simple 2D games: twin stick shooters and tower defence.
Twin stick shooter games as the name implies, involves an input device with two analogue joysticks, such as a modern Xbox controller. One of the sticks moves the player, while the other stick lets the player aim. Usually, the aim of the game is to destroy surrounding targets to increase your score until you are hit yourself.
Tower defence games on the other hand involve the idea of waves of enemies with the intent of destroying a base located on a top-down map. The player(s) must then place down turrets or other obstacles to help obstruct the enemies' goal of reaching the base. This can be done with placed turrets

which shoot at the enemies, or walls which force them to take longer routes. These are afforded by destroying previous enemies.

The final decision was to have a game which took inspiration from those two genres and Wii Play Tanks as mentioned earlier. The game features either one or two players controlling their own tank using controls similar to that of a twin stick shooter. The controller of choice will be an Xbox One controller as that is the available equipment to hand. The left analogue stick will control the movement while right analogue stick aims the canon on top of the tank. Pressing the trigger will cause a shell to be fired from the canon and can bounce off objects a few times before being destroyed. Enemy tanks will then spawn from the edge of the playing area and approach the player's base in the centre of the screen.
The player tanks can place down walls which will force the enemy tanks to change their route towards the base and buy the players extra time. Enemy tanks are destroyed by either direct fire from player tanks, or by reaching the player's base which will cause the player's base to be destroyed.
There will be at least two different types of enemy. One enemy will not feature any offensive weapon and its only goal will be to reach the base. The other enemy will feature a canon, similar to the player's which can fire back towards player tanks.
If an enemy tank is destroyed by the player, the player's score is increased, and there is a chance of it dropping a coin. The player can pick the coin up which will allow them to spend money on walls. Money is shared between the players so coordination is necessary to build an effective defence.
If a player tank is destroyed, it is removed from the field (game) while it is repaired, in which it will return after a short amount of time has elapsed. However, if the base is destroyed, the game ends and the player's score is presented.

1.1.2 Why Not Use a Game Engine?

The pros of using a game engine is that a lot of complicated functions and code are already written for you and all you need to do is call those relevant functions. Of course, this does not make making a game easy, it simply takes away some of the tedious work involved with writing standard functions over and over. Game engines also provide a form of abstraction where games are clearly, and often visually broken down into individual parts such as sprites, scripts, and models.
Another reason for using a game engine is that they often provide tools for quickly creating parts of the game, such as the ability to drag and drop elements to form game levels or to form user interfaces by placing buttons or other visual elements. Furthermore, by using a well-known engine, you can be assured that the provided tools and code has been thoroughly tested so there is less testing involved compared to starting from scratch. The advantages continue if creating a 3D game becomes the subject as these engines often already support rendering of 3D models with complex lighting, texturing and shading algorithms included. However, I will not be creating a 3D game for my major project as this would be too ambitious to complete in the allocated time and resources. It is important to note that this is not an exhaustive list of advantages to using a game engine but rather a brief summary of some of the biggest reasons. Of course, I am looking at this from the perspective of an individual making a game, whereas studios will have entirely different reasons for choosing to use one engine over the other or not using a third party one at all.

On the other hand, there are reasons as to why my project will not involve the use of a game engine. One main reason is that most popular game engines are closed source and often hide the underlying workings to the game developer. While this can be useful as it is a form of encapsulation and abstraction, it does mean that a game developer does not have full control over the exact architecture and design of the game. However, this is not true for every game engine, for example, one relatively

recent game engine “Godot” (Linietsky & Manzur, 2007-2022) is completely open source unlike engines such as Unity. Nevertheless, it can be very difficult to completely understand and modify the underlying engine code. Another key advantage is that most game engines are not free in terms of license or cost. For example, Unity costs no money for an individual to start using the engine, however, as soon as you want to properly publish a game and remove the branding, it begins to cost a lot of money and you are still restricted on how you are allowed to distribute your game.

In conclusion I am not using a game engine so I can better understand the underlying workings of my game, to reduce unnecessary performance overhead and to ensure I have full rights over my game. This is largely possible because of the limited scope of my game and the fact that I will not be using 3D components.

1.2 Analysis

After developing a clear idea as to how the game would work, further research was required to work out what technologies and programming languages would be used to create the game. Clearly the most important aspect would be the platform, followed by language, followed by available libraries or tools for that given language.

1.2.1 Choosing a Platform

The platform of choice was quickly selected as the game needed to be run on something accessible and easily available to many people, including myself. Therefore, deciding to use a desktop computer/ PC as the platform of choice seemed to be an intuitive decision. However, choosing an operating system to focus on was more involved. The two considered options were Microsoft Windows 10(+) or Debian 11 – Linux. The benefits of choosing Windows over Linux is that it is more commonly installed and readily available. According to W3Schools (Refsnes Data, 2022), 66.3% of their visitors used Windows 10 to access their website in February of 2022 compared to 4.1% with Linux.
However, the benefit of focusing development on Linux is that Linux is free and open-source which aligns with the same reason for not using a proprietary game engine. It also means that the barrier of entry to install Linux on a machine is indeed lower than that to install Windows.
In conclusion, it was decided that the game will prioritise targeting Windows due to its greater support for various libraries and accessing (Microsoft) Xbox One controllers is typically easier to do on Windows due to the inclusion of the XInput API. However, while development will focus on Windows, if possible, the game will target both Windows and Linux support depending on the technologies used.

1.2.2 Researching the Right Programming Language and Libraries

In terms of programming language, two options were investigated. The first option researched, was C++ which was primarily chosen to due to its prevalence in the games industry. According to mooc.org (edX, 2021) “C++ is the most popular language for creating game engines” and its prevalence is evident as soon as you research the production of large commercial games. Therefore, the big benefit of using C++ is that a lot of valuable experience could be gained from using the language to design a game from scratch. This experience could be valuable when potentially looking for a job in the near future. C++ is often chosen for game development due to its speed and efficiency. Due to its low level and compiled nature, C++ code is executed much faster than Python for example which is interpreted. However, this advantage is not necessarily important as the game is simple in nature and should be able to run on any modern computer regardless of the CPU, GPU or RAM inside.

Of course, C++ is a low-level language and therefore a library would be needed to perform basic expected tasks such as drawing shapes, text, sprites to the screen, loading and display of image files and taking controller inputs.

Two readily available libraries were already known due to research in December 2021, and these were SDL (Simple DirectMedia Layer) (Sam Lantinga and SDL Community, 2022) and SFML (Simple Fast Multimedia Library) (Gomila, 2018) .

Both of these libraries are similar and generally provide a comparable list of features. They can both print images to the screen, render text, draw lines and basic shapes, and take inputs from a controller. One key difference is that SFML is easier to begin with and is built with object-oriented designs in mind, whereas SDL is more C based. During this period, it was easy to follow SFML documentation on the official website to set up a simple project and this provided confidence that there were plenty of resources to help. Furthermore, one of the biggest advantages of using these libraries is that they offer cross platform support which would allow both Windows and Linux platforms to be supported as discussed as a potential goal earlier.

SFML was chosen over SDL due to its simplicity and easy to follow documentation, not to mention its object-oriented focus which would save time. SFML provides additional features such as audio and networking APIs, however there are no plans to use these, although it does provide the opportunity to add features in the future.

However, early on in planning and discussions with my supervisor, it became clear that another language may be a better choice. This was mostly due to experience and proficiency. The low level advantages are not necessary for a simple 2D game and therefore, while using C++ would be a good option, it may not have been realistic to use an entirely new library to make a game of this size in the allocated time.

Therefore, HTML5 and JavaScript was considered as a likely replacement. JavaScript appeared to be a much more viable option due to plenty of previous experience and therefore a much better option to reduce risk. A big advantage of using JavaScript is that it is generally hardware agnostic and can run on anything which has a modern internet browser. JavaScript is also a higher-level language compared to C++ which would mean that designing the game would be slightly easier since pointers, type specification and garbage collection would not be required and is done automatically. Additionally, many modern browsers feature debugging tools which can allow for easy and quick bug fixing during the testing periods.

However, a potential caveat of using JavaScript would be the possibility of gaining controller inputs since this is not a feature which JavaScript was originally intended for. Fortunately, though, modern browsers such as Firefox and Chromium have added a GamePad API which allows access to multiple simultaneous game controller inputs. Additionally, JavaScript has not always had object orientation as a built-in feature, however, since 2015, it has been added to the language specification and thus modern browsers.
The introduction of HTML5 also provides the canvas tag which is used to draw graphics similar to that of the previously discussed C++ libraries. The Mozilla Developer Network (Mozilla Foundation, 1998-2022) has a wealth of documentation on this feature which will be referenced continually throughout development.

Given that HTML5 and JavaScript provided all the required features while reducing the risk posed with the C++ route, it was decided that JavaScript would be used to implement the game.

1.2.3 Risk Analysis

In section 1.3 each feature has been assessed based on the risk involved in designing and implementing it. A higher risk means that the feature is likely difficult to implement or is critical to accomplishing the goals of this project. Should a risky feature fail to be implemented, large sections of the game may need to be reconsidered. Therefore, features which pose a higher level of risk are completed as early as possible.

1.3 Process

1.3.1 Methodology

An agile based feature- driven development methodology will be adopted during this project. Using FDD helps to structure the design and development as it breaks the solution down into individual modules which can be individually implemented. This is well suited for a game which is made up of various classes which each perform their own task. Of course, this is a solo project however, so features will not be divided up as they would in a team and will be instead designed and developed in sequence.

Features are implemented in individual iterations which have a specified goal. Each iteration in this document will consist of three stages: design, implementation and testing before a new iteration then begins which may depend on the outcome on the previous iteration. For example, an unforeseen issue may be discovered which may need addressing in the following iteration.
Figure 1 shows the sections inside each iteration.

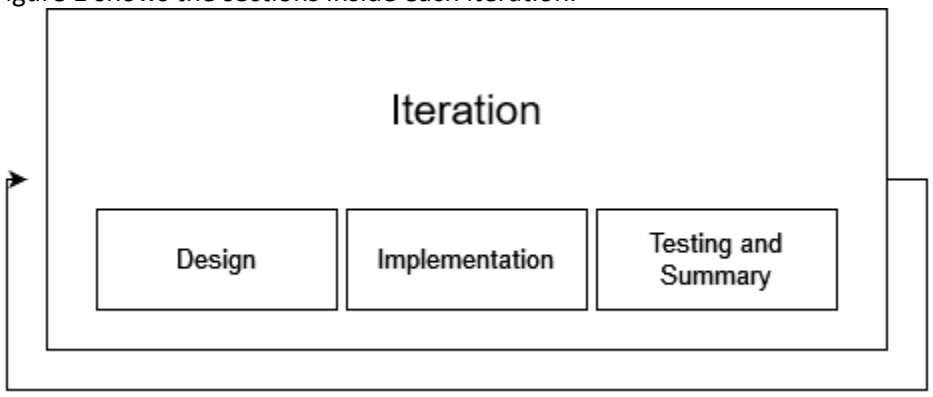

Figure 1 - The basic sections a single iteration will contain.

1.3.2 Design Section

The design sections in each iteration discuss the design decisions and considerations made while planning how the specified feature were to be implemented. Any complex issues is researched and discussed in this section along with potential issues and a solution. The first iteration has a large emphasis on design as the overall architecture, as well as how each subsequent component will fit will be discussed first. Following iterations will have a more narrow focus, mainly on the specific feature that is being worked on.

1.3.3 Implementation Section

The implementation sections discuss how the design translated into code. It focuses on potential problem areas which occurred and how they were overcome.

1.3.4 Testing and Overview Section

This section briefly explains the tests executed on the code introduced in the current iteration. If a test fails, it will be discussed why said test failed and how it could be fixed, or if following iterations need to be tailored to help fix a major issue discovered in the last iteration.

1.3.5 Proposed Features in chronological order of Intended Implementation

Feature Number	Relevant Iteration	Description	Assessed Risk (0-10)
1	0	Implementing boiler plate code and basic game loop.	1
2	1	The player's base which appears in the centre of the map.	1
3	1	Taking inputs from gamepad to control player tanks movement and aiming.	6
4	1	Implementing a virtual grid to place game entities.	2
5	2	Player can place down walls which do not obstruct the enemies.	1
6	2	Collision detection is implemented.	3
7	2	Shells can be fired from player tank and bounce off walls.	5
8	3	Enemies can automatically route a path to the players base.	9
9	3	Striker enemies can fire shells at the player.	4
10	4	The game has a menu system which can be navigated by using the controller.	2
11	4	Enemies are spawned automatically by difficulty determined by the current score.	3
12	4	The game can be paused.	2
13	4	The game ends when the base loses all its health and the score is presented on game over.	1

2. FDD Iterative Stages

Due to following a Feature-Driven-Development methodology, Design, Implementation and Testing sections are combined in each iteration rather than having their own larger sections. The hope here is that the game can be clearly broken down into modules and by developing in iterations, it will keep the focus in one area.

2.1 Iteration 0 – Initial Design and Setup

As discussed in the process section, the first iteration of the project has a large emphasis on design. This is to allow for the creation of the "base" of the game in which following features can be integrated into. How the project was setup and what tools are used will also be discussed.

The goal of the following section is to describe the overall architecture of the game which explains how all the subsequent features are added, and other designs which were considered.

2.1.1 Overall Architecture Design

Component Based or Object Oriented

In game design, two types of overall designs are typically discussed, these are component based and objected oriented. Object oriented is often seen as the standard, old fashioned way of structuring a solution to a game. These approaches are good because, much like other applications, they effectively structure the application into distinct and separate objects which allows for code re-use and reduces coupling. For example, different types of enemy can inherit common movement code from a parent enemy node so that the child nodes do not get too complex or contain repeated code. However, as games got more complex over time, so did the variety of different classes and objects with the need to share code. This often meant that base classes easily got overwhelming and thus a new design emerged.

Game engines such as Unity and Godot have resorted to using a component-based game design. Components are individual pieces of functionality (classes) which are connected to other game objects. They are unaware of other game components until they are needed. This makes the design very modular as a component can be removed and the rest of the game object will still work.

The architecture for this game will take an object-oriented approach. While component-based architectures are common in larger games and engines, an object-oriented design is often much easier to implement for smaller games and does not carry the overhead of having to implement a component system. Additionally, the game is not large enough to warrant the use of a component system as many components would only be used once anyway thus some of the benefit of reducing code complexity would be lost.

Limitations of JavaScript

Another important aspect which impacts the rest of the design is the JavaScript language itself and its limitations. While today, JavaScript is often used to make web games, that was not its original purpose and this has meant that some important features such as Classes, which are standard in other languages, work slightly differently in JavaScript. For example, Java applications have an entry point

inside a main class, however, standard JavaScript applications must first start in a procedural way and then instantiate its own class.

Additionally, classes as a feature was only introduced in 2015 and until very recently, it was not possible to specify public and private fields within classes. Unfortunately, this feature was only integrated into Firefox in version 90 (June 2021) which was missed in the latest Debian Linux release which means these features cannot be used for this project. This is notable because some aspects of the design, such as the class diagram featured in chapter 3 reference encapsulation on methods and variables which could not be physically implemented into the code due to this limitation.

Abstract Game Objects

The game is of course made up of various objects such as player tanks, enemy tanks, and walls among others. Since an object-oriented approach has been decided upon for this project, it makes sense that there is an abstract base class which will then have child classes which implement specific features and components such as player tanks and shells.

Common Methods

Therefore, any visual element on screen will inherit from a base “entity” class which will have a common structure so that common game loop functions can be called. The base Entity class has two methods which must be overridden by child classes. The first is the “tick” method which will cause that specific game object to run one cycle, i.e. a player may move one unit. The other is the “draw” method which runs code which draws that current game object to the HTML canvas.

Update Rate/ Frame Rate

The consequence of this design means that the draw method can be called at any rate, for example, 60 times per second for 60 frames per second. However, the tick function must be called at a constant rate, such as 30 frames per second, if this rate changes, the game will speed up and down accordingly.

This means that the tick rate was decided early and kept constant throughout development so that all game objects run to the same time scale. For example, after programming the player movement, the tick rate should not change otherwise the enemy movement may be programmed with shorter intervals in mind which would then make the player move slower.

The industry standard refresh rate which is achievable while also “believable” is 30 frames per second. This is an ideal target because anything less can make the individual frames of animation quite apparent which can be distracting to the player. Going much higher means that you must ensure the game is efficient enough to reach that target otherwise you may get a situation where the computer is not able to keep up, and therefore the method is no longer called at the expected rate which causes the game to actually slow down.

Common Variables

The abstract Entity class will also define which data is required for it to be a valid entity object. Since entities will always be a visual object, they must have an x and y coordinate so that it can be drawn to the correct position on the screen. Similarly, it must have a width and height so that is drawn to the screen correctly. However, most game objects have a tile-based design which means they have the same dimensions (32px).

Entity Object Container

Once an entity object has been instantiated, it is pushed to the entity array in no particular order. A function within the “main” JavaScript file then loops through the entity array 30 times per second (every 33.3 ms) calling the tick method of each object to update itself, followed by the draw method, to display on the screen.

Game Screen

The game is tile based, where each tile is a square forming a grid which forms the game map. This means the player has a top-down birds eye view of the playing area similar to that of a traditional board game.

Due to hardware limitations, tile-based games traditional have tile sprites with dimensions in multiples of 8. This game is made with tiles of 32 pixels squared as this provides a balance between simplicity yet showing enough detail so that a player can quickly understand what something is.

The initial design was to have a map 16x16 tiles in size which means the game canvas would be 512 pixels squared. This is big enough to see while small enough to fit on any screen size, such as a laptop and provides plenty of space for the player to place down walls, thus allowing enough strategy.

Figure A (below) shows an illustration made early in late February to illustrate the approximate intended look of the game. Visible elements include the player’s base surround by walls, a green player tank and an incoming enemy tank.

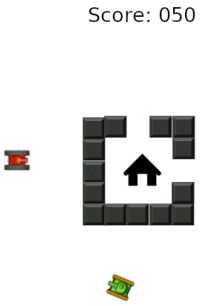

Figure A

Use of IDE/ Text Editor

JavaScript is an interpreted language which means an IDE is not needed to compile the code or to run debugging tools. Therefore, simple text editing tools are appropriate as to provide the most screen space, free from unneeded buttons and features.

On Windows 10, Notepad++ was used which has syntax highlighting and features to collapse functions to simplify looking at larger game object classes. On Debian, gedit was used which also features basic syntax highlighting.

Diagrams

The web app, draw.io (JGraph Ltd, 2021) has been used to produce all of the diagrams contained within this document.

2.1.2 Implementation

This iteration was mostly a design phase and thus only a small amount of code was implemented.

The index.html page was created with links to the aforementioned main JavaScript file which waits until the page is loaded before running the initial method. The initialise method simply creates a new JavaScript interval which calls the game loop function every 33.3ms. This loop function, loops through the whole entities array as explained in the design.

One minor decision was made to implement the canvas tag in the HTML file as opposed to creating it with JavaScript, which simplifies the main JavaScript file a bit. The downside is that this approach makes the game slightly less modular as it now means that the main JavaScript file requires there to be an existing canvas element in the HTML file otherwise it will not run.

Keeping Track of the Canvas

One other issue which had not been accounted for was passing a reference of the canvas element to the draw method in each game object entity. A webpage can have multiple canvas elements and therefore without a reference, the method has no way of drawing images or shapes to the screen.

Dynamically accessing the HTML body and checking for a canvas was considered as this would have been modular but would have also had a large performance impact since it would be called multiple times per second. Therefore, it was decided that the CTX variable, which is a reference to the canvas should be passed as a parameter to the method instead. This variable would later get combined inside the gameData object.

2.1.3 Testing and Overview

The browser of choice used to run the game on each system was Firefox Version 97.0.2 for Windows 10 and Firefox ESR 91.6.0 on Debian 11. However, the game was developed using standard JavaScript features as found on Mozilla's MDN (Mozilla Developer Network) (Mozilla Foundation, 1998-2022) website which is cross compatible with all modern browsers (from 2016).

Firefox, among other browsers include helpful debugging tools such as a console where lines can be printed out to help test code. More importantly however, code can be injected into the page to run functions manually which is a useful tool to test functions and features which have not or cannot be implemented into the main game loop at that time.

The use of informal testing should not be underestimated throughout the development of the game. Manually running the code after each small addition and checking the console to ensure that no obvious errors had occurred really helped ensure that implementation got completed as smoothly as possible without running into large bugs only encountered after significant amounts of code had already been written.

At the end of each iteration, a set of tests are conducted to ensure that the goals of the iteration have been completed and to make sure that it is of a suitable quality. The test tables can be found in Appendix A.

Note: referencing chapter 3 of this document may help in the understanding of how the individual iterations were integrated.

2.2 Iteration 1

The main goals of this iteration were to display the base, player tank and collect gamepad inputs from the browser so that the player can move and aim. This was being completed early on as collecting data from gamepads was identified as a potentially risky task.

2.2.1 Design

Visual Design and Theme

This is the first iteration in which sprites are used, therefore meaning a common theme was established so that the games sprites work together and do not look out of place. Because the tiles are small and only a few pixels in size, it was decided that Piskel (Community, 2021) would be used to produce the sprites. Piskel is a web application which allows you to make pixel style animations or sprite sheets. The app also has tools to allow you to format sprite sheets in a specific dimension so that it works for many projects.

It was decided that the game would feature few, and flat colours which looks good while being easy to produce. Game objects have a dark outline so that they can be easily identified by the player.

Creating a Base Class

The base class is very simple since its constructor only needs an x and y value which will always be the centre of the map. The base is also a standard tile which means its dimensions are the selected 32 pixels width and height.
The Base class extends Entity; however, the tick function was not overridden since the base does not have any logic. At least for now, it just remains in the centre of the map.

Creating a Player Class

This section begins with the task of designing the player class so that individual player tanks can be controlled. Player of course, extends the base abstract Entity class which was implemented in the previous iteration. This means that the Constructor, Tick and Draw functions must be overridden so that specific player functions can be implemented.

The Constructor

Players require three values to be instantiate: an x and y coordinate along with a player number. This is important as the player number ensures the tanks are visually distinct. Depending on the player number, a different tank sprite is displayed which allows players to quickly tell which one they are controlling.

The player number variable also controls which connected controller that class will read inputs from, so only one controller can control a single tank.

Updating Tick

The tick function first needs to get the left analogue stick values from the correct controller. The x and y coordinates of the analogue stick can then be multiplied up and then added onto the existing x and y

coordinates which should cause the player to move. When the stick is released, the stick's coordinates should switch back to 0,0 so no movement will be applied to the player.

A visible reticle moves (aims) and responds based on where the players right analogue stick is. Getting the reticle to move is relatively simple since it is very similar to processing player movement. The right stick's x and y position is just added on top of the players current position. However, this would not be enough information to process the rotation of the player tank's canon using the canvas rotate function which requires an angle. Thankfully, the stick's x and y position can be aligned to the adjacent and opposite line of a triangle which means that the inverse tan in trigonometry can be used to calculate the angle to rotate the tanks canon. However, either an if statement or Math.atan2 would need to be used to prevent a division by zero when the y axis on the stick is moved but x is exactly zero.
The tanks rotation angle is stored in a variable so that the draw function can read it once it is called.

Updating Draw

The draw function first needs to get a reference to the two player tanks sprites and use the correct one depending on whether the player was instantiated as player one or two. The tanks lower section will be drawn first which does not rotate. The higher section of the tank, the canon will be drawn on top of the base. The canvas rotate function is used along with the angle processed in the tick function to rotate the turrets angle to the correct position, which should directly correspond the right thumb stick's position. The aiming reticle is then drawn last and its x and y position is relative to the players tank rather than the whole game window. Therefore, if the tick function calculated the reticle's x value as 32, that would be 32 pixels to the right of the player, and not the window.

2.2.2 Implementation

Map Size Issues

When implementing the base, it quickly became apparent that because the map had 16x16 tiles, there would not be a centre tile to place the base. Thankfully, the simple solution was to simply increase the map size by 1 on each axes. Therefore, the game window is now (32x17) 544 pixels square.
It was during this period that it was also realised that a scale multiplier maybe useful to integrate into the draw methods so that the game window can be made larger or smaller. However, this is not an important feature so it will not be tested regularly.

Problems with Tank Canon Rotation

Getting the player tank's turret to rotate to the correct angle using Math.atan(theta) only seemed to work when the x axis on the analogue stick was positive. When the stick got pushed to the left, the canon would snap back to the right side, so it would never point to the left. Discussing the issue with a close friend highlighted that the issue was that the inverse tan would only provide a value between 0-180. Therefore, an if statement was added to check if the x value is negative (pointing to left), in which Pi (180) degrees is added to the angle.

Combining Both Player Tank Sprites

While producing the player tank sprite sheets in Piskel, it became quickly apparent that it would be a much better idea to have both tanks in the same image on top of each other. Therefore, the player class needs only access one image and then offset the y axis if the tank is player two.

Implementing a Deadzone to Fix Stick Drift

When the left analogue stick was released, it would go back to the centre. However, due to imprecision and potentially cost saving measures in the production of the controllers, the axes values do not return to zero, but rather a very low noisy number. This meant that the player tank would slowly creep towards one direction of the screen when it should be stationary.

The fix required calculating the magnitude of the left analogue stick and comparing this to a constant value in a variable which could be adjusted. If the magnitude of the stick values was below the threshold, the movement scripts would not be run. This was also applied to the right stick so that the aim would become more stable.

Concerns Regarding Dividing by Zero

The noisy output from the gamepad controllers meant that it was realistically impossible to have a situation where a value was divided by zero when using the inverse tangent.

2.2.3 Testing and Overview

Major Cross Compatibility Issue

Design and implementation of the player tank class was done on Windows 10. However, when testing the code on Linux, it was immediately clear that there was a compatibility issue. Using the left analogue stick caused the tank to move as expected, however, the right stick's x axis cause the aim to adjust, while one of the triggers on the controller controlled the other axis, thus aiming was incredibly difficult and completely unintended.

There was not much information regarding this issue online but is likely due to a lack of Direct X/ xInput support on Linux, due to the software and APIs being proprietary. Therefore, the current design of getting controller inputs directly via the web gamePads API was not going to work if Linux and Windows were to be supported. One solution would be to drop cross-compatibility across different operating systems. However, this was a core design decision, therefore it was decided that the next iteration should focus on writing a GamePad Controller which would be an interface between the hardware, browser, and game.

2.3 Iteration 1.1

Iteration 1.1 was unexpected until the testing stage of the previous iteration. The main problem involves different controller mappings on Windows and Linux. Therefore, the goals of this iteration were to implement a GamepadController class which would translate calls from the game so that returned values were consistent regardless of the operating system. Notice how figure 2 and 3 highlight the differences in number of axes and buttons detected.

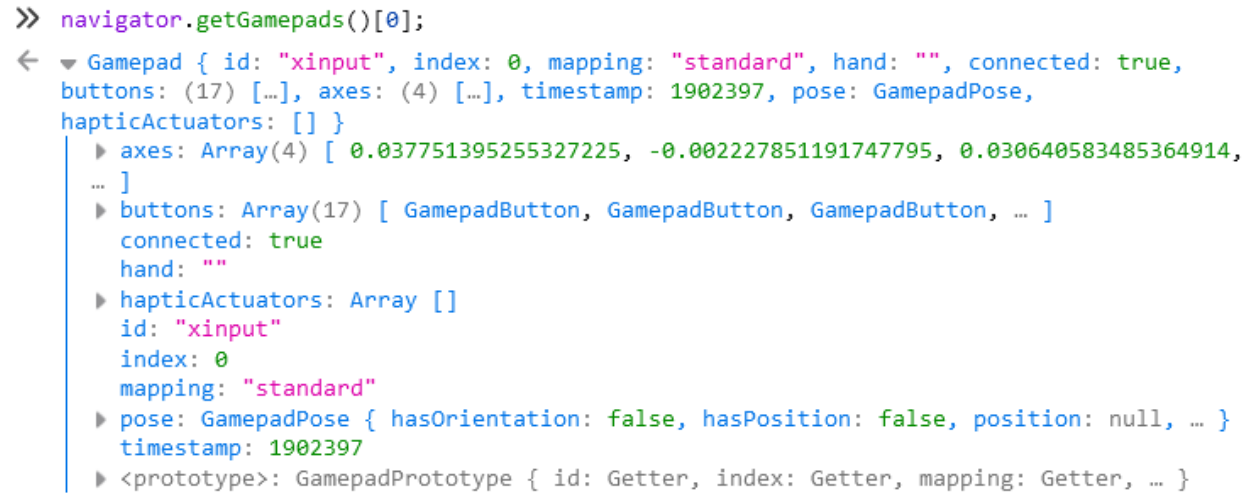

Figure 2 - Windows gamepad object

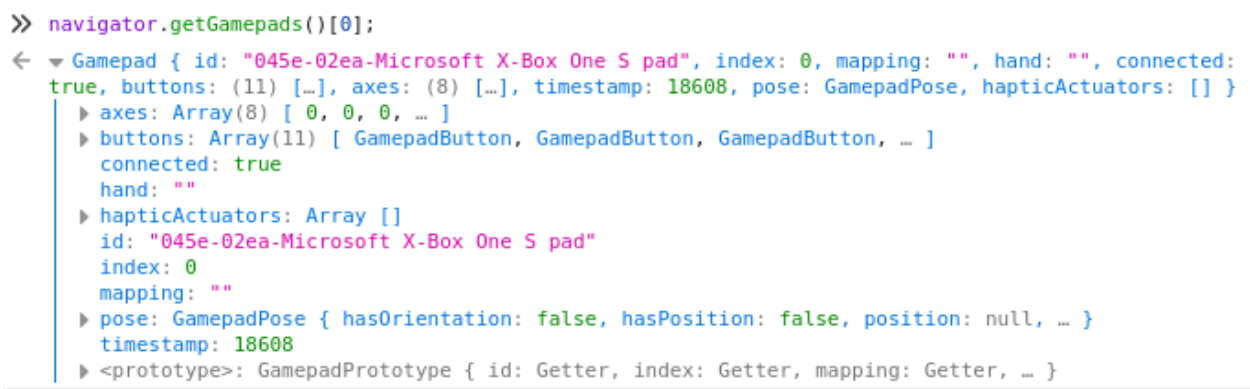

Figure 3 - Linux gamepad object

2.3.1 Design

Purpose

The GamepadController class sits between the game and browser. Game objects such as the player make a call to the gamepad controller class which then queries the browser depending on which platform is running (i.e. Windows or Linux). It will then return a consistent response so the rest of the game is not concerned with the mappings of the controller, which is handled separately by this class. The diagram (figure 4) below shows the difference:

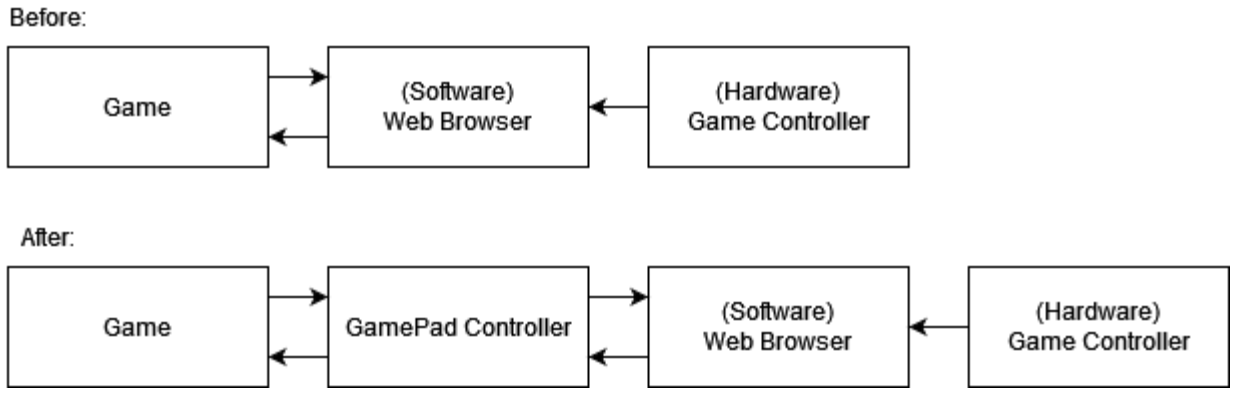

Figure 4

Vibration Support

During research into reading data from the gamepad, it was discovered that some browsers support vibration feedback. This is a feature that was considered, but not integrated at this stage because it was a non-standard feature and not essential to the game.

Integration into Existing Code

The gamepad controller class is instantiated in the main JavaScript file when the software first initialises. Two Boolean values are taken as parameters; one to enable printing to the console, for debugging purposes, and the other to enable non-standard features such as vibration feedback which would be implemented at a later stage.

The main JavaScript file contains the gameData object which contains data important to every part of the program, such as references to the canvas, and future game related variables such as shared player coins. The GameData object stores the gamecontroller instance so it can be accessed by the various game object entities. Game entities receive a reference to the GameData object via parameter in the tick and draw methods.

Required Methods

The web browser GamePad API breaks the game controller down into various parts such as buttons, triggers, and joysticks. Therefore, it is logical to bring the same separation into the gamepad controller via different methods for accessing different parts.

getStick(index, side)

Get stick returns the x and y coordinates of the specified side, on the specified controller.

getTrigger(index, side)

Get trigger returns an object based on that found in the standard browser API, which contains three variable states: pressed, touched and value. This will be used in the coming iterations to allow the player to fire shells using the right trigger, although left trigger readings are included so that the code could be reused for future purposes which may involve the use of the left trigger.

getButton(index, buttonName)

This method contains a constant which stores button mappings for both platforms and returns the correct button given the provided button name. This form of abstraction also makes the code more readable since buttons are identified by name rather than an integer.

```
» gamepadController.getButton(0,"A");
← ▸ GamepadButton { pressed: true, touched: true, value: 1 }
» navigator.getGamepads()[0].buttons[0];
← ▸ GamepadButton { pressed: true, touched: true, value: 1 }
```

Figure 5 – A snippet of code in the console showing how the status of the "A" button can be returned

Figure 5 shows how the new implementation on line 1 is far more intuitive to understand than the previous standard API version below it.

Converting to xInput (Windows) Format

The outputs of the gamecontroller class more closely resemble that of the Windows format. This is because a quick investigation showed that Firefox, Edge, and Samsung Internet on Android all have a similar set-up and therefore Firefox for Debian was the outlier. Going this route also meant that a new format would not need to be created and this new class could easily be integrated into existing code, such as the Player class.

2.3.2 Implementation

Implementation involved creating preliminary methods to test the functionality before adding a full working implementation.

Adding a Method to Check for Operating System

Checking for the correct button mapping was going to happen inside the get functions, embedded as a variable set by a ternary operator. However, a refactoring period which occurred shortly after the preliminary implementation caused this functionality to be moved to a separate function: isStandard thus increasing code reusability and increasing maintainability.

The OS check used to be achieved by checking for the presence of the word "xinput" in the id string. However, this proved unreliable as a quick test on Microsoft Edge failed the check and the game thought that it was running on the Linux mapping. Therefore, the function now checks for the number of analogue stick axes values, which is the biggest difference between the two button mappings.

Converting Digital to Analogue

One of the biggest differences between the Windows API and the Linux API is that Linux treats the left and right D-pad as one axes in an analogue fashion represented by a floating-point number. I.e. right d-pad provides the value 1, while left d-pad provides the value -1. This is also true for the up and down buttons which also have their own axes attribute.

On the other hand, the Windows API treats these buttons like any other buttons on the controller. Therefore, on Linux it was necessary to manually create an object identical to one which would have been returned on Windows. An if statement is used to check the value of the floating-point variable which would then decide the state of the Boolean value which would be returned to the game.

2.3.3 Testing and Overview

Frequent informal tests were run throughout implementation to check that each stick and button was being correctly captured. At the end, manual testing was performed to ensure that all the required inputs had been captured and were being returned correctly. These tests can be found in Appendix A.

Once the new code had been integrated into the existing code, functionality of the player was tested once again to ensure that nothing had broken or changed. This testing was performed on Firefox ESR on Debian 11 and Firefox on Windows to ensure that tank behaviour is identical. This was also the first time that the tank was fully functional on Linux.

Future classes and functionality would now only use this new class to communicate with the controller as that would ensure that the game runs on both Windows and Linux devices. Other devices and platforms would almost certainly work too, however these will not be tested as frequently as testing multiple browsers would be very time consuming.

2.4 Iteration 2

The main goals of this iteration were to add walls that the player tanks can place on the map which would also coincide with the introduction of collision detection which ensures that the player cannot pass through the new walls or the existing base.
Bullets/ shells are also developed in this iteration. Shells should bounce off walls and the canvas edge but not tanks or the base.

2.4.1 Design

Tile Based Placement of Walls

It was already decided in the first design iteration that game objects would each be a consistent 32pixel wide square. However, players can move gradually between squares rather than moving one square instantly. Therefore, it was time to weigh up the decision of being able to place walls anywhere, or being able to place them only on a 17x17 grid.
It was decided that walls would be placed on a set 17x17 grid as this would make it much easier for the player to quickly place walls during the game rather than struggle with precision placement. Furthermore, and more importantly, it would make it much easier later in production to calculate paths for the enemies to follow if walls are placed on a strict grid. It is possible to calculate navigation routes for A-Star without a grid, however it would be much more complex.

A Virtual or Explicit Grid?

One key design decision here focused on how the placement of walls would be implemented. One option considered was having a 2D array which would be the same size as the game map. Each unit in the array would contain a wall object which contains information about that particular space such as if there is a wall there, and if so how much health it has. The potential benefit of this approach is that it would be impossible to have walls placed in invalid spaces however, it would mean that there would be a lot of additional data being stored and would also be more difficult to integrate into the current entity rendering system.

It was decided that a virtual grid should be used instead. This means that the player would still only be able to place walls in specific allocated places, however these would not be contained in a 2D array. Wall objects will have their own x and y positions and will get pushed onto the entity array like any other object. This increases efficiency while making for a much simpler design however, it does mean that walls could be placed in unintended places, such as between tiles.

Collision Detection

Collision detection is handled within the base Entity class since multiple game objects will need to check if they are colliding, such as player tanks and enemies with walls. The collision detection function will take a point and loop through the Entity array to ensure that it does not hit a solid game object. This function can be called for each side of a tile to work out if a tile can move in a certain direction or not. This is different to a box collider which would determine a collision, however working out which side would be slightly more difficult.

Keeping Track of Shells

Tank shells were originally designed so that they would be contained inside an array of the game object that fired them. This would mean that it would be easy to determine what tank the shell was from and

therefore, how many times it would bounce. Player shells should bounce twice whereas enemy shells should not bounce. However, a problem with this design is that it would mean that once an enemy is killed and removed from the game, their shells would instantly disappear. Therefore, enemy shells are their own game object entity contained within the large entity array. They will take a parameter once instantiated which will dictate how many times they will bounce. Once a shell has reached its bounce limit, it will remove itself from the array. If a tank is destroyed while its shells are currently in game, they will only be removed once they destroy themselves.

2.4.2 Implementation

Using an Array to Keep Track of Collisions

During implementation stage it was discovered that checking collision points of each side of the tile would not be enough since collisions on the corners would not be detected. Additionally, placing the collision points on the corners of the object would mean that the direction of the collision could not easily be determined. For example, a collision in the top right corner could either mean something is on the top or right side of the object. Therefore, 8 points of collision, two on each side were checked and an array of four Boolean values would be returned. The four values would correspond to whether movement was allowed up, down, left or to the right.

Implementing an Entity ID system

During the implementation of shells, it was discovered that shells would instantly hit the tank that instantiated them, as they would be inside the collision region. This meant that an id would need to be stored of the object that created it. However, there currently is not a variable assigned to an entity which can uniquely identify it. Therefore, a static id counter was created in the abstract entity class which would keep track of what number it had assigned previous instantiations and provide a new unique identity each time a new game object Entity is created.
Once this system had been created, the shell entity would now take an additional attribute on instantiation which would be the ID of the tank which created it. Using this ID, along with a JavaScript Date object, a tank would be immune to its own fired shell for a few milliseconds just to give enough time for the shell to move out of its hitbox.

2.4.3 Testing and Overview

Testing the bullets was more challenging since there can be a huge amount of variability in which a bullet can hit an object. However, the final manual tests found in Appendix A ensure normal and extreme cases were tested so that bullets should work within normal game conditions.

Once this section was complete, the game was beginning to look more like a game with projectiles moving across the screen automatically. However, risky sections would still need to be designed and implemented and these would be focused on next.

2.5 Iteration 3

The main goal of this section was to focus on identified risky, and thus complex sections such as the A-Star path finding algorithm and the addition of different enemy types.

2.5.1 Design

How Enemies and Pathfinding Class Interact

The two different enemy tanks “Streaker” and “Striker” extend common functionality found in a generic abstract base class called “Enemy” which extends the Entity class. Common functionality deals with the movement of enemy tank towards the base as both enemies do this continuously. The path class is separate and does not extend any other class. Each enemy will instantiate its own path object which will perform its pathfinding needs and will also store the resulting path which the enemy will follow. Figure 6 shows an outline of the relationship.

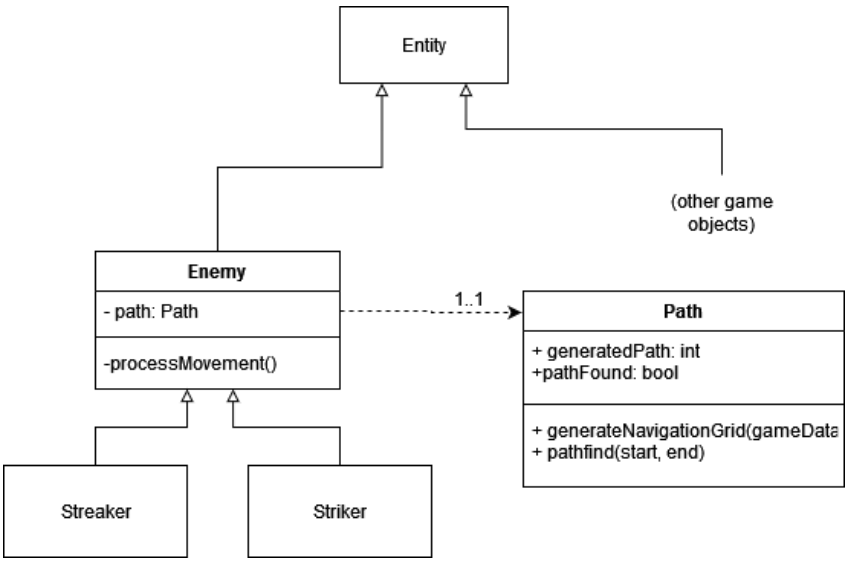

Figure 6

Generating a Navigation Grid for Pathfinding

Before a path can be found from the enemy to the player base, a grid must be generated which contains data about all of the tiles within the game.
The data about each tile will be:

- The x position of the tile
- The y position of the tile
- Whether there is a wall there (i.e. traversable)
- gCost, hCost and fCost which is the distance from the start, to the end, and both summed up respectively
- The tile’s origin which will chain together tiles to form a path once the pathfinding algorithm completes.
- Whether the tile is a start or end point

Of course, not all of these values are calculated immediately, but they are set up so that the A-Star pathfinding algorithm can use them to calculate the shortest path. At this earlier stage, the navigation generation simply loops through all possible tile locations to check if there is a wall there. If there is, it will mark that space as being non-traversable. An alternative design was considered which would have instead looped through the entities array checking for wall presence and marking spaces as non-traversable if one were present. However, this approach would have been slightly less efficient, and the selected approach was more intuitive and easier to understand when reading the code.

Enemy AI

David Hunter asked during the mid-project demonstration whether enemies would use a state machine or a different AI model. After consideration it was decided that a state machine would not be necessary since there are only two states. These two states also only affect the striker enemy because it indicates whether the enemy tank is shooting at the player tank or not. A state machine is not necessary as there is no complex change of strategy such as tanks going from moving to stationary or stationary to shooting. Enemy tanks will never be in a situation in which they cannot move since the player is prevented from completely closing off the route to the base.

2.5.2 Implementation

Increasing the Size of the Navigation Grid

During the implementation stage it was decided that the pathfinding algorithm would be more robust if the navigation grid was surrounded with a wall of untraversable tiles. The reason for this is that it would prevent the pathfinding algorithm trying to check tiles outside the bounds of the map which could result in crashes. However, this addition meant that the size of the navigation grid was now larger than the game map which means that the coordinates between the two would need to be converted before the resulting path is processed by the enemy path AI.

Added Code to Prevent Game Crashes During Testing

The pathfinding loop is by far the most complex and biggest loop implemented yet. It is contained within a while loop and should only exit once the destination is reached or all nodes are processed and considered. However, during testing the game would often crash which would cause the browser to freeze and the debugging console to break. It was determined that the freezing was occurring due to the pathfinding running in an infinite loop, therefore a loop counter was added to ensure the loop would exit even if it would never normally have ended. This allowed the debugging tools to keep working and thus the issue was fixed.

Enemies Hitting Walls and the Base

One thing which was not considered when designing the Enemy AI, which became a problem was dealing with what happens when an enemy hits a wall. Once the code had been implemented and tested, it was brought to attention that enemies would not re-route if they are blocked by a wall which was placed after their initial pathfinding run had completed. Therefore, an if statement had to be added which forces a new path to be generated.

Additionally, it was not previously considered that the base was a solid object, and therefore the enemies would be blocked from entering its area. Therefore, collision detection had to be disabled for their last movements.

2.5.3 Testing and Overview

Testing the pathfinding algorithm was rather difficult due to all the possible permutations and variability. However, both simplistic and complex paths were tested which helps ensure that players are unlikely to construct mazes which have not been tested.

The console print function in the pathfinding class paid off during the early stages of implementation where it was used extensively to test the pathfinding algorithm before enemies were implemented. This meant that the path class could be fully tested and functional before the implementation of enemies began.

Test tables for this iteration can be found in Appendix A.

2.6 Iteration 4

The main goals of this iteration were to implement the user interface and add the main game loop with enemies spawning automatically. Coins and explosion effects were also added in this iteration.

2.6.1 Design

The Different UI States

The game has four main states: main-menu, playing, paused and gameover. The transition between the states are described in the state transition diagram below in figure 7.

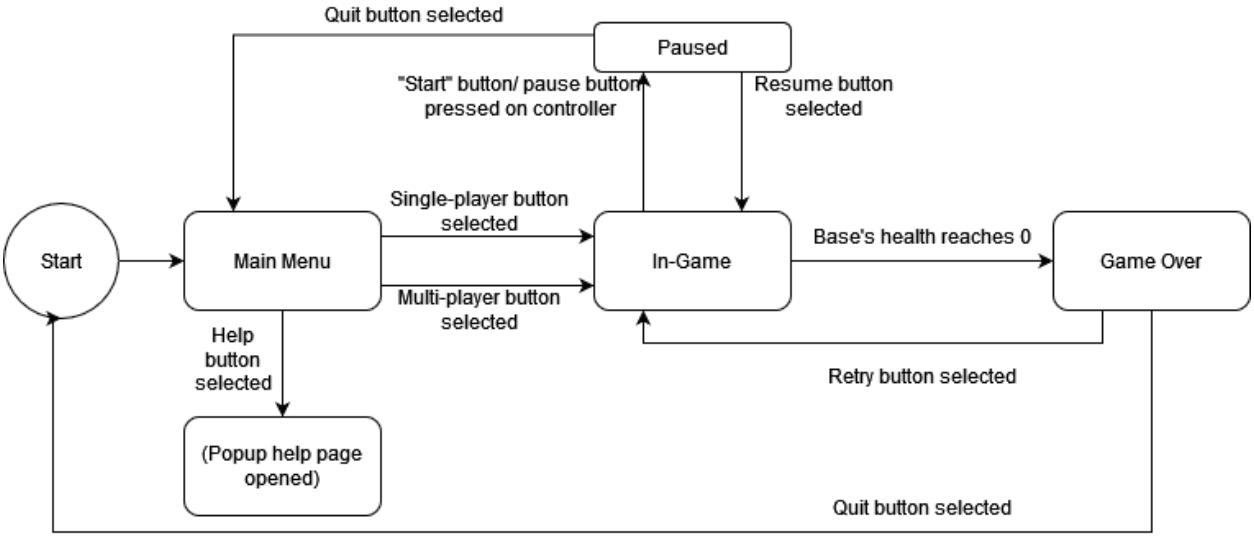

Figure 7

Integrating the UI

The user interface was not considered during the preliminary design stages and therefore, deciding on how it was integrated into the current code was considered in this iteration. One approach which was considered during the earlier stages was to have the UI element classes, such as buttons, as game object entities, much like other parts of the game. This would have made it simple to integrate without creating new classes. However, this idea was not finalised as this would have made it possible for UI elements to have been obscured by recently created game objects as there is no z-index ordering when drawing to the screen. Additionally, it would have been cumbersome to add code which has to search through the entity list to remove and add relevant UI components as they would not necessarily be grouped together.

Therefore, it was decided that game states should be contained within classes which are known as managers. These is similar to game managers within the Unity Engine which means that they govern the overall flow and control of the main game loop.

Main Menu Manager

The main menu manager takes inputs from the first connected controller which allows player one to select buttons on the main menu: single player, multi-player, and help. When the game is selected, the

main menu manager is unreferenced, and the game manager is swapped in. The main menu also displays the number of connected controllers which will help players debug potential problems with controller connection issues.

Game Manager

The game manager is instantiated once the game begins by the main menu manager. The game manager handles the main game loop from start to end. This means that it automatically spawns enemies into the game and keeps track of the base's health to ensure that the game should continue. Once the game ends, the global state variable will change, thus updating of the game objects will cease, and a "game over" overlay will appear showing the player's score with two buttons to quit or retry. The game manager also draws the UI on top of the current game after all the other game objects have been drawn. This includes the health bars of the players, current score, coin count and base health.

Enemy Spawning

Enemies spawn using pre-programmed difficulty stages which are invisible to the player but are selected in the background by the game manager. The pre-programmed difficulty stages are contained within an array, with each entry being a small object which contains three values. The first value specifies at what player score the difficulty should apply at. If there is a difficulty level above the current above, they are checked too to ensure that one should not be selected instead. The second value states what interval the enemies should be spawned in milliseconds and the third states the possibility between 0 and 1 in which a striker enemy will spawn instead of a Streaker.
Once the spawn interval has passed, the timer resets, and an enemy is spawned at a random position (using Math.random) along the game screen's perimeter. The player class is updated so that they are unable to build walls along the perimeter thus blocking the enemy from spawning.

Difficulty Balancing

One of the simplest yet most effective way at balancing the game between single player and multi-player was by spawning the same number of tanks, of which there are players every time the spawn interval has passed.

2.6.2 Implementation

Incompatible Pause Mechanics

Before the pause menu was implemented, the ability to pause the game was tested by using the console. Most of the functionality worked however, while everything seemed to have stopped, once the game had resumed some changes would appear immediately after resume when they should have not. For example, an enemy would instantly spawn after resuming the game even if one had recently been spawned. It was discovered that the reason for this was because much of the games timing was calculated using JavaScript's Date object which uses real world time. Of course, it is impossible to freeze real world time so therefore some changes apply immediately after resuming even if they should not have. The only solution that seemed possible was to rewrite much of the game's timing scripts, however this was an unrealistic target at this late stage, thus the pause function was dropped.

2.6.3 Testing and Overview

Continually running informal tests during the development of the final iteration ensured that some unusual and unpredictable bugs were found. For example, coins can build up during busy areas of the map, and in the right circumstances may mean that shells can pass through enemy tanks leaving them unharmed. This is thought to be possible because the current collision detection can only detect one collision instance at a time and by chance this could end up being the coin rather than the tank. An easy solution would be to allow the enemy tanks to “collect” coins however this change was not implemented.

During this stage it was also found that the main menu could be skipped altogether accidentally if the user presses the “A” button to connect the controller. To fix this, a delay was implemented so inputs are only taken from the controller after 200ms since the last button press.

3 Design Summary

This section contains general information about the complete design of the game after all the iterations were finished. It may be helpful to refer to this section while reading through the iterative stages, or while performing maintenance.

3.1 Entity UML Class Diagram

Figure 8 (below) shows the relationship between all of game Entities. User interface components such as screen managers are not included.

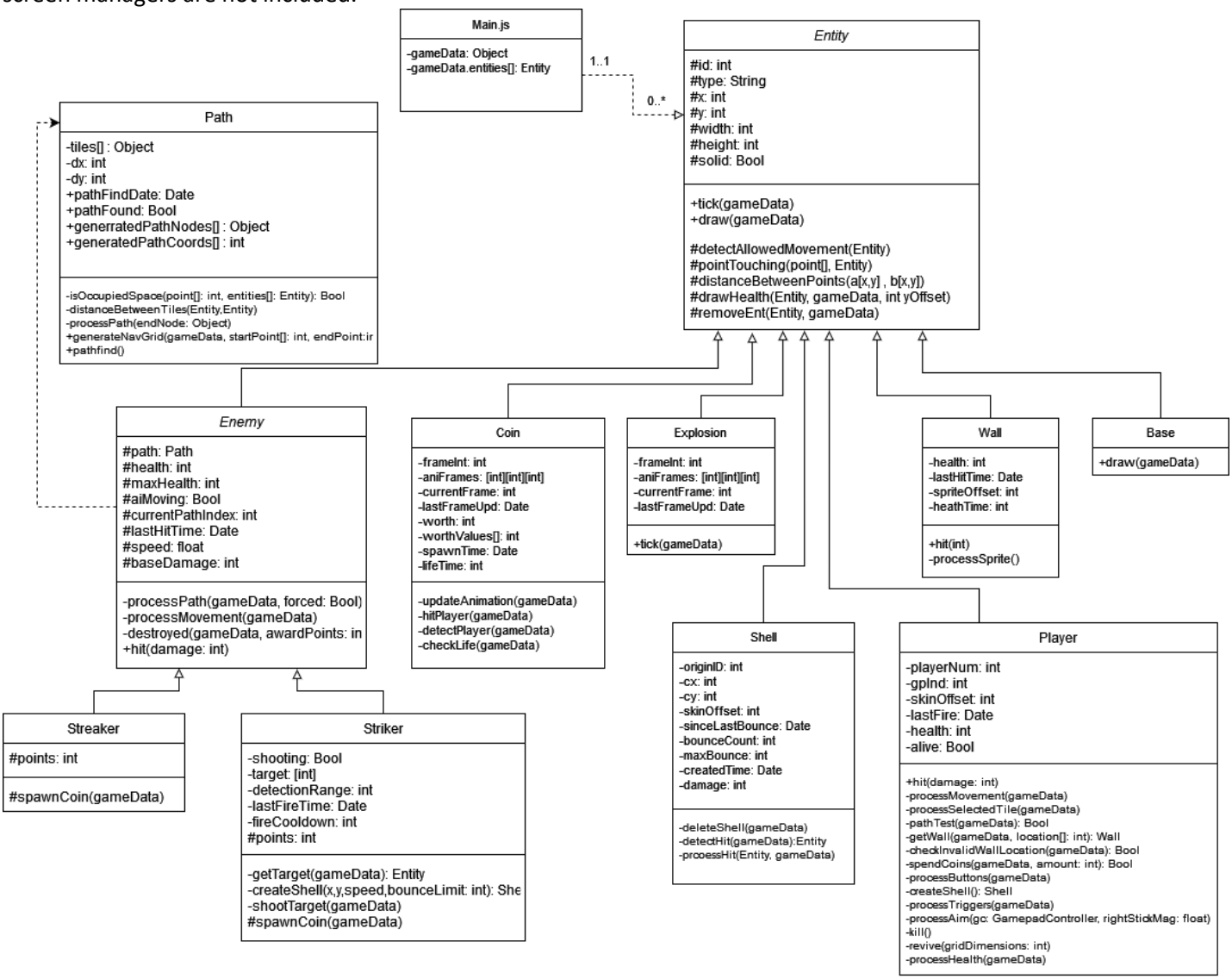

Figure 8 - Class Diagram

3.2 Brief Class Descriptions

Main.js
This is not a class but rather the procedural entry point of the code. It contains the gameData object which contains all the data about the running game. It also contains a reference to the current screen manager as discussed in section 2.6

Entity
Entity is an abstract class for game objects, not including user interface components. It has common methods which many subclasses will routinely use and standard (tick and draw) methods which need to be overridden by the subclasses before they can be implemented.

Player
The player class draws the player tanks to the screen. It also handles movement and aiming from controller inputs. This class also handles the players placing walls and checks to see if they are in valid locations before instantiation. Player bullets are also instantiated from this class.

Shell
The shell takes a direction once instantiated and processes and draws the movement of one shell. It detects hits and processes bounces. Once it has detected a hit, it will invoke the hit method of the game object it has collided with (if it has such method) along with the damage done as an integer.

Base
This class is simply a placeholder for the base and does not actually calculate or process base health, which is contained within the gameData object. This class draws the base to the screen every frame and prevents players and bullets from passing through it.

Wall
This class represents a single wall instance which keeps track of how much health it has and draws a wall sprite in its location.

Path
This contains information and methods relevant to calculating a path from a start position to an end position on the board. It is used by enemies to move them towards the base, and by the player class when creating walls to ensure that they do not block access to the base.

Enemy
This abstract class handles common functionality such as movement of the enemies.

Streaker
This class extends enemy and integrates functionality specific to the Streaker, such as the chance to drop a silver coin on destruction and scoring the player 100 points.

Striker
The Striker class handles the shooting behaviour which is exclusive to this enemy type.

GamepadController

Handles platform differences between Windows and Linux when getting gamepad inputs.

GameScreen.js

This class is the screen manager running during the playing of the game which handles enemy spawning and game over window.

TitleScreen.js

This class is the screen manager running once the game is first opened. It manages the main menu buttons and connected controller icons.

Ui-button.js

This class contains code related to drawing and displaying buttons used in the user interface.

4. Critical Evaluation

This chapter will weigh up the successes and shortcomings of the project including aspects that could have been done differently.

4.1 Tool Use

Text Editor

Overall, I believe that my choice to use Notepad++ and gedit as text editors for developing the code were sufficient. Popular alternatives include Microsoft Visual Studio Code which is cross platform and contains a plethora of additional features such as add-ons, preview windows and other extensions. Many of these features were not essential and would possibly take up screen space better used for viewing the code. However, it may have been worth further investigating Visual Studio Code, especially during the latter stages as some refactoring tools would have been quite helpful, especially when it comes to renaming variables spanning across different files.

Version Control

One area in which not enough attention was focused in was file structuring and version control. The use of git was not considered until late on, however this was something which should have been discussed and considered by iteration 0 at the latest. Using a git service such as GitLab would have provided me a central location whereby I could have controlled updates pushed to the code. Additionally, GitLab would have provided issue tracking tools which I also did not discuss in this report. Issues were instead tracked manually on paper which would not be appropriate had this project been developed as a team.

For this project I stored the source files in a Dropbox folder which was installed on a Windows and Linux machine. This allowed me to rapidly protype code without the need to manually push and commit to a repository. However, this drawback could have been mitigated by using tools with built in git support such as the aforementioned Visual Studio Code text editor.

Internet Browser

I chose to use Firefox to help test and develop my game. This was primarily because it is cross platform and contains useful developer tools such as a console which was used extensively throughout development to help test sections which were not fully integrated into the game yet. For example, I used the console to manually instantiate new enemies before that was done automatically in the game loop.

Ultimately however, the choice of browser did not have a significant impact on the development of the game. Most modern browsers all have a similar level of tools available, however Firefox was the most accessible.

4.2 Game Design

Technical Design

Overall, I believe that the design of my game was generally well thought out, especially at the beginning stages and that meant the game was divided into well selected features which could be developed piece by piece and then easily integrated together. For example, once I had designed the Entity system as discussed in iteration 0, I then knew the general structure of every game object, such as its required parameters and methods to override. This meant that once a class had been implemented, it could be

instantiated and then pushed to the Entity list where it would then run automatically and appear in the game.

However, the design was not completely robust, and some aspects could have been improved. For example, the user interface was seen as low risk, and this meant that it was given little attention until the last iteration. This was when it was discovered that the current designed had not considered how adding the user interface would change the structure of the code. In the earlier stages, it was decided that UI elements could be implemented like any other game object. However, once it was discovered that this was not an ideal solution, it was too late to make large changes to the code to facilitate an ideal structure, therefore some compromises were made. For example, there is a global gameData variable which holds the array of game objects despite the fact that game objects should only be visible during the presence of the game screen manager, thus the ideal placement of this array should have been inside that class.

Gameplay

While not the core focus of this document, I believe that the resulting game is of a high quality and is genuinely very fun to play. The controls are intuitive and easy to pick up for someone who has at least some prior experience with a game controller. The difficulty curve as time goes on gives the player plenty of time to set up their walls before large waves of enemies appear, thus there is a large amount of strategy and potential playstyles that people can adopt. The gameplay also adjusts as time progresses. For example, during the earlier stages of the game, only streakers appear which do not feature any offensive weapons. However, later on Strikers appear with canons which means that players feel an increasing need to defend themselves in addition to their base which results in the player potentially adopting a different technique in order to get a high score.

Were all the Features Implemented?

Overall, I believe that not only did the game meet the requirements of the feature list, but also surpassed them. All the key features such as local multiplayer, enemy AI and pathfinding were all implemented within the deadline which were all indicated as having a certain degree of risk involved. Additionally, non-essential features were also added to improve the quality of the game such as explosion animations and two different coin types.

4.3 Process and Choice of Methodology

I believe that the choice to adopt an agile based feature driven development methodology has significantly increased the quality of the work and reduced risks compared to a traditional plan-based approach such as a waterfall method. This methodology was adjusted slightly to suit the needs of this project, for example, by removing parts which would allocate work to different people in a team. Using an agile approach allowed me to adjust the requirements of subsequent iterations based on the findings of the previous iterations. For example, I was able to quickly identify that controllers behave different on Linux than they do on Windows, therefore, the next iteration was adjusted so that an additional class could be developed which helped me handle inputs on both Windows and Linux. If I had adopted a plan based design, it would have only been after the whole game had been designed that this issue would have been encountered and this could have made adjustment very difficult and time consuming since the design would need to be changed throughout to accommodate an additional class.

Adopting an FDD methodology also allowed me to correctly identify the required classes and objects for the game. This is evident as the resulting gameplay matches the early design sketches and descriptions I had produced.

4.4 Future Improvements

If development of this game were to continue into the future, I would like to make some additions to the game and changes to the development process. Future work could include:

- Refactoring the code to improve the design of the game. For example, the entity array could be moved into the gameplay manager as game elements should not be present on any other screen.
- Completing the title screen with game logo.
- Ensure the latest browser version is installed on each platform so that new technologies can be utilised to improve design. For example, class access modifiers are a new technology which could be integrated into the current design so that code is better encapsulated.
- A better collision detection algorithm could be implemented which adapts better to game objects which are slightly smaller than a tile such as tanks.
- A sound managing class could be implemented which would allow for there to be sounds effects and music playing in the background.
- A settings menu could also allow players to enable and disable certain features. For example, players could choose to enable or disable controller vibrations, change the zoom on the screen or to change buttons mappings.
- The source code should be uploaded to a version control system so that issues can be more effectively tracked and so that the code can be more easily worked on as a team.

These changes would increase the quality of the game and make it of a standard more in common with commercially available games which could be sold on online storefronts.

4.5 Report

Writing reports is something I find personally difficult. This report has been no different and while I believe is does convey important decisions taken and potential alternatives, it also misses a lot of key details. This is possibly due to less time allocated to planning the report which has resulted in a less than ideal document.

The structure breaks the main development cycle down into iterations, much like the development which allows the attention to be focused on specific features of the game at any one time. I believe that this is an improvement over having one initial design section and one large implementation. However, some of the content could have been moved to earlier sections which would have made parts of my project easier to understand.

Overall, I believe the report is of acceptable quality, however additional time and planning would have made it easier to understand for someone who has not previously seen the code. A perfect report would highlight the high standards of the technical work completed however I do not believe that this report demonstrates this as well as it could have done.

4.6 Shortcomings

One of the major issues with the final game is that it is not yet complete and ready to be played by a general audience. This is because of few missing key features such as a complete title-screen with game logo as shown in figure 9 (below). This is partially due to the late decision of the game's name: Tank-off! Creating a high-quality art piece was seen as a low priority and thus cut to allow for more important features to be refined.
Another feature which was missed was a pause menu which is basic functionality expected from any modern game. As discussed in iteration 4, this feature was cut due to a prior design decision which based some game timings on real-world time. Therefore, this is a clear indication that if I were to complete this project again, I would spend more upfront resources in allocated specific target dates for each iteration and feature to be completed by.

Another lesson learned from completing this project is to employ a greater level of research when designing tests for future projects. Spending more time in the earlier stages researching potential ways to automate tests can pay off in the long run. For example, having at least some form of automated game testing would have saved a lot of time manually testing, especially towards the end of the project where all the different classes get integrated together into the final game. Additionally, test stubs could have been used to provide perfect input values from a virtual game controller so that tests could be perfectly replicated rather than done manually by hand which can introduce variability. Test stubs for the gamepad controller would have also allowed me to input extreme values which would be very difficult/ realistically impossible for a real user to accomplish yet could occur, such as a precise 0 value on one analogue stick axes.

One issue that the game currently has is that game physics and movement are tied directly to the frame rate/ tick rate of the game. This means that should the frame rate be increased, the game's mechanics would break rendering the game unplayable. Therefore, a "delta-time" variable should have been included which normalises the game object's movements based on the time taken to render the previous frame therefore producing a stable movement regardless of update rate.

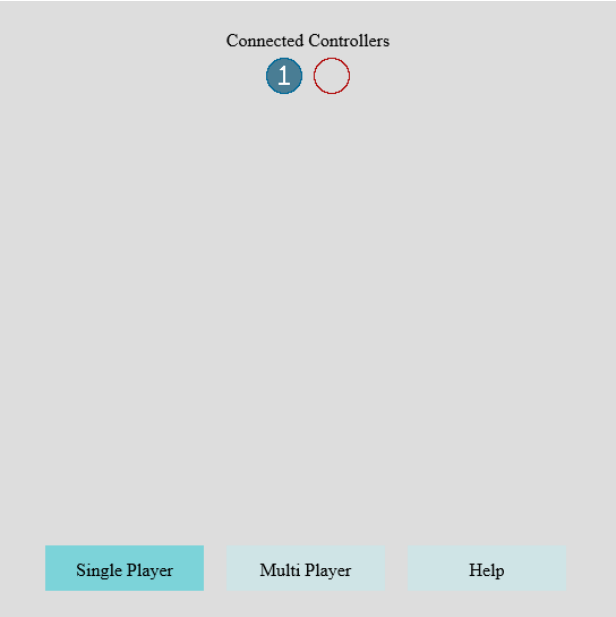

Figure 9

4.7 Conclusion

In conclusion I am very pleased with the output of my technical work. I am especially pleased that all of the assets were produced by myself and that no game engine was used which means that there are no third-party licenses which restrict the distribution of the game.

However, I am slightly disappointed with the result of the report as I believe this could negatively impact the perceived quality of the work accomplished. If I had a chance to complete the project again, I would allocate more time to plan the report so that it more effectively conveys the development of the game.

However, in conclusion I have ended up producing a game which is very fun to play and has achieved all of its major feature requirements therefore I believe that this project has been a success even if there are aspects which I would change on a second attempt.

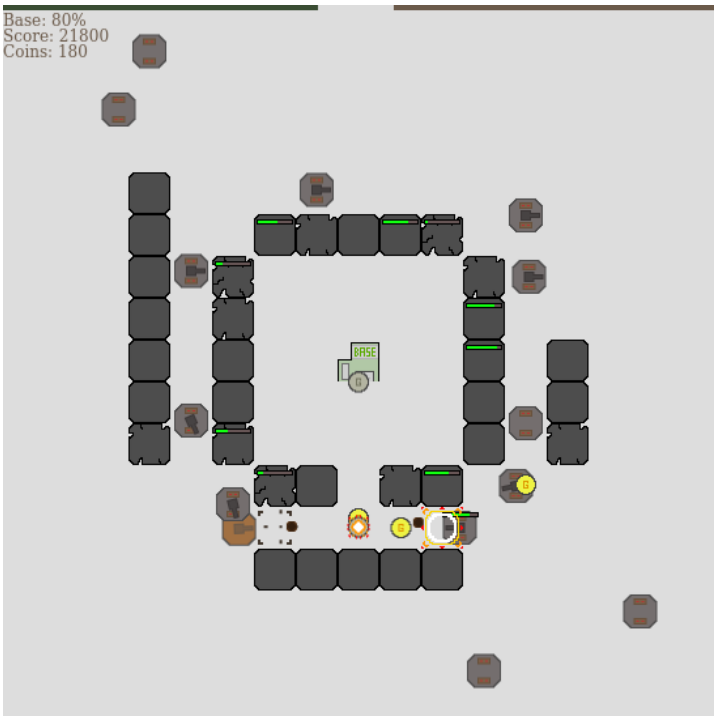

Figure 10 - A screenshot of gameplay

5. References

Community, 2021. *Piskel.* [Online]
Available at: https://www.piskelapp.com/
[Accessed 26th April 2022].

edX, 2021. *Best Programming Languages for Game Development.* [Online]
Available at: https://www.mooc.org/blog/best-programming-languages-for-game-development
[Accessed 24 April 2022].

Epic Games, Inc, 2004-2022. *Unreal Engine.* [Online]
Available at: https://www.unrealengine.com
[Accessed 5th May 2022].

Gomila, L., 2018. *Simple and Fast Multimedia Library.* [Online]
Available at: https://www.sfml-dev.org
[Accessed 13th December 2021].

JGraph Ltd, 2021. *drawio.* [Online]
Available at: draw.io
[Accessed 24th April 2022].

Linietsky, J. & Manzur, A., 2007-2022. *Godot Engine.* [Online]
Available at: https://godotengine.org/
[Accessed 5th May 2022].

Mozilla Foundation, 1998-2022. *MDN Web Docs.* [Online]
Available at: https://developer.mozilla.org/en-US/
[Accessed 28th April 2022].

Refsnes Data, 2022. *OS Platform Statistics - W3Schools.* [Online]
Available at: https://www.w3schools.com/browsers/browsers_os.asp
[Accessed 25th May 2022].

Sam Lantinga and SDL Community, 2022. *Simple DirectMedia Layer.* [Online]
Available at: https://www.libsdl.org/
[Accessed 13th December 2021].

Unity Technologies, 2022. *Unity Real-Time Development Platform.* [Online]
Available at: https://unity.com/
[Accessed 5th May 2022].

Valve Corporation, 2015. *Steam Link.* [Online]
Available at: https://store.steampowered.com/app/353380/Steam_Link/
[Accessed 4th May 2022].

6. Appendices

Appendix A – Test Tables

Iteration 1 Goals and Testing:

Iteration Goals:

1. The player can be moved using the left analogue stick on the controller
2. The player can aim using the right control stick on the controller
3. The base is displayed in the centre of the map

Test ID	Relevant Goal	Description	Expected Outcome	Pass/ Actual Outcome
1	1	Hold left stick to the left.	Player tank should move to the left of the screen.	Pass
2	1	Hold left stick to the right.	Player tank should move towards to right.	Pass
3	1	Hold left stick downwards.	Player tank should move down the screen.	Pass
4	1	Hold left stick upwards.	Player tank should move up the screen.	Pass
5	1	Hold left stick up and to the right.	Player tank should move diagonally up and to the right.	Pass
6	1	Push left stick gently towards the right.	Player tank should move slowly towards the right.	Pass
7	2	Push right stick towards the right.	Tank aiming reticle should appear on the right side of the tank.	Pass
8	2	Push right stick towards the left.	Tank aiming reticle should appear on the left side of the tank.	Pass
9	2	Push right stick upwards.	Tank aiming reticle should appear above the tank.	Pass
10	2	Push right stick downwards.	Tank aiming reticle should appear below the tank.	Pass
11	2	Push right stick to the top left corner.	Tank canon should point towards top left corner of the screen.	Pass
12	1	Push left stick fully to the right and then release to let stick flick back to centre.	The tank should move to the right and then stop once released.	Pass
13	2	Push right stick fully to the right and then release to let stick flick back to centre.	The aiming reticle should point to the right and remain once released.	Pass

Iteration 1.1 Goals and Testing

Iteration Goals

1. Button inputs can be read from the controller on Windows and Linux
2. Trigger values can be read from the controller on Windows and Linux
3. Analogue stick values can be read from the controller on Windows and Linux

ID	Relevant Goal	Description	Expected Output	Pass/ Actual Output
1	1	Use the browser console and an instance of the gamepad controller class to get inputs of all the following buttons whilst the buttons are pressed: A, B, X, Y, Up, Down, Left, Right, Left Bumper, Right Bumper, Start, Select	The code returns an object containing a “pressed” value which has the value **true** for each corresponding button.	Pass
2	1	Use the browser console and an instance of the gamepad controller class to get inputs of all the following buttons whilst the buttons are not pressed: A, B, X, Y, Up, Down, Left, Right, Left Bumper, Right Bumper, Start, Select	The code returns an object containing a “pressed” value which has the value **false** for each corresponding button.	Pass
3	2	Use the console to get the value of the right trigger when not pressed.	The returned value should be 0.	Pass
4	2	Use the console to get the value of the right trigger when half pressed.	The returned value should be approximately 0.5	Pass
5	2	Use the console to get the value of the right trigger when fully pressed.	The returned value should be 1.	Pass
6	2	Use the console to get the value of the left trigger when fully pressed.	The returned value should be 0.	Pass
7	2	Use the console to get the value of the left trigger when half pressed.	The returned value should be approximately 0.5.	Pass
8	2	Use the console to get the value of the left trigger when fully pressed.	The returned value should be 1.	Pass
9	3	Press the left and right analogue stick towards the right and use the console to return the X axis value.	The returned value should be approximately 1.	Pass

10	3	Press the left and right analogue stick towards the left and use the console to return the X axis value.	The returned value should be approximately -1.	Pass
11	3	Press the left and right analogue stick upwards and use the console to return the Y axis value.	The returned value should be approximately -1.	Pass
12	3	Press the left and right analogue stick downwards and use the console to return the Y axis value.	The returned value should be approximately 1.	Pass

Iteration 2 Goals and Testing

Key Goals:

1. Walls can be placed.
2. Players cannot pass through walls or off screen.
3. Bullets can be fired by the player.
4. Bullets bounce off walls but not bases or other players.

ID	Relevant Goal	Description	Expected Outcome	Pass/ Actual Outcome
1	1	Place tank tile cursor over a tile on the far-right border and click “LB” to build a wall.	Wall is not built – cannot build on boundary.	Pass
2	1	Place tank tile cursor over a tile closest to the base and click “LB” to build a wall.	Wall is not built – cannot build on boundary.	Pass
3	1	Place tank cursor over valid building location and click “LB” to build a wall.	Wall is built in selected tile.	Pass
4	1	Place tank cursor over existing wall and click “LB” to build a wall.	Wall is not built on existing wall tile.	Pass
5	1	Place tank cursor over existing wall and click “RB” to destroy a wall.	Wall is removed from the game.	Pass
6	2	Move player to furthest right possible towards the edge of the screen.	The player should not go off screen.	Pass
7	2	Move player to furthest left possible	The player should	Pass

		towards the edge of the screen.	not go off screen.	
8	2	Move player to furthest point upwards towards the edge of the screen.	The player should not go off screen.	Pass
9	2	Move player to furthest point downwards towards the edge of the screen.	The player should not go off screen.	Pass
10	2	Place a wall on the map and then starting from below, move the tank upwards into the wall.	The player should collide with the wall and stay below it.	Pass
11	2	Place a wall on the map and then starting from the right side, move the tank towards the left into the wall.	The player should collide with the wall and stay on the right side.	Pass
12	2	Place a wall on the map and then starting the the left side, move the tank towards the right into the wall.	The player should collide with the wall and stay on the left side.	Pass
13	2	Place a wall on the map and then starting from above, move the tank downwards into the wall.	The player should collide with the wall and stay above .	Pass
14	3	Aim the player's turret towards the left and press the "RT" button on the controller.	A shell is fired from the tank and travels west.	Pass
15	3	Aim the player's turret north and press and hold the "RT" button for approximately three seconds.	The tank should fire shells which head north for approximately three seconds and then stop	Pass
16	4	Aim the player's turret towards a screen boundary and pressed the "RT" button.	The tank's shell should bounce off the first wall, off the second and then disappear	Pass

			after hitting the third wall.	
17	4	Connect a second controller and aim the second tanks turret towards the first player. Press the “RT” button to fire a shell.	The shell should immediately disappear at contact with a tank.	Pass
18	4	Aim the tank’s turret towards the player’s base and press the “RT” trigger to fire a shell.	The shell should immediately disappear at contact with base.	Pass
19	4	Place a wall on the map and aim the player’s turret towards the wall. Press the “RT” trigger to fire a shell.	The shell hitting the wall should reduce the walls health.	Pass

Iteration 3 Goals and Testing

Main Goals:

1. Pathfinding class can find routes from A to B (edge of screen to centre base)
2. Streaker enemies continually move towards base and detonate upon impact
3. Strikers have same functionality as Streaker but they also have a turret which shoots at the closest player within range.

ID	Relevant Goal	Descriptions	Expected Outcome	Pass/ Actual Outcome
1	1	Place down walls surrounding the players base and instantiate a path instance using the console. Call generateNavGrid() with the starting point being the top left corner and the destination being the base. Then call the consolePrintGrid function.	The map should be shown in ASCII format with the end point denoted by an “E” and surrounded by walls denoted with “[X]”. An “S” should denote the start in the top left corner which is surrounded by walls.	Pass
2	1	Repeat test 1 except call the pathfind function after the navigation grid has been generated. End by calling the console print function.	No path should have been found so the outcome should be the same as the test 1.	Pass
3	1	Surround the base with walls in a “U” formation which has been	A path should have been found which reaches around the “U”	Pass

		rotated clockwise by 90 degrees. Generate the nav grid with the parameters used from test 1. Then call the pathfinding function to generate a path. Finally, call the console print function to show the results.	formation and back in towards the base.	
4	2	Use the console to spawn a Streaker at any edge location.	The enemy should automatically route itself towards the base and disappear once the enemy is fully on top of the base.	Pass
5	2	Surround the base in the same wall formation as described in test 3. Spawn a Streaker enemy on the left side of the map.	The enemy should automatically move around the wall and then back in towards the base where it disappears.	Pass
6	2	Spawn a Streaker anywhere on the edge of the map. Aim the player's turret towards the enemy and press the fire button 5 times.	The enemy's health should reduce after each shot and then disappear after the fifth shot. The enemy should not disappear any earlier and no bullets should go through them.	Pass
7	3	Spawn a Striker anywhere on the screen boundary. Slowly move the player towards the enemy until they start shooting. If the player is already within shooting distance, gradually move distance away until the enemy stops shooting.	The enemy should not shoot while the player is far away. Once the player is within distance, the enemy should fire every 1.5 seconds.	Pass
8	3	Spawn a Striker anywhere on the screen boundary. Move player 2 close to the Striker and then move player 1 within striking distance but not as close as player 2.	The Striker should always target the closest player and therefore, player 2's health should be reducing after each shot received.	Pass

Iteration 4 Goals and Testing

Main goals:

1. Title screen appears on start with functional display and buttons.
2. Enemies spawn automatically during game.
3. Enemies explode upon death and sometimes drop coins.
4. Heads up Display is visible during game.
5. Players die and respawn.
6. Game over screen appears at end with functional buttons.

ID	Relevant Goal	Description	Expected Outcome	Pass/ Actual Outcome
1	1	Start game by opening index.html in web browser.	Title screen appears which shows connected controller icons and “Singleplayer”, “Multiplayer” and “Help” buttons appear at the bottom in which the “Singleplayer” button is already highlighted.	Pass
2	1	Start game and connect one controller. Press the “A” button on the controller.	Title screen appears which shows one controller connected.	Pass
3	1	Start game and connect two controllers. Press the “A” button on both controllers.	Title screen appears which shows two controllers connected.	Pass
4	1	Start game and press left d-pad button on the controller.	“Single player” button remains selected.	Pass
5	1	Start game and press right d-pad button on the controller twice.	“Help” button is highlighted.	Pass
6	1	Start game and press right d-pad at least three times.	“Help” button is highlighted.	Pass
7	1	Start game with one controller connected and then select “multi-player” button.	Gameplay does not start.	Pass
8	1	Start game with one controller connected and then select “single-player” button.	Game screen changes and one player tank appears.	Pass
9	1	Start game with two controllers and then select “multi-player” button.	Game screen changes and one player tank appears.	Pass
10	2	Start the game and wait 7 seconds.	One enemy should appear.	Pass
11	2	Start a multiplayer game and wait 7 seconds.	Two enemies should appear.	Pass
12	2	Start a game and ensure the score is at least 3000, keep playing until score reaches 9000.	At least one striker should spawn.	Pass
13	2	Start a game and check spawning locations for at least 30 seconds.	Enemies should spawn randomly from each 4 sides of the map.	Pass
14	2	Start a game and check to see if spawn interval decreases by one second after gaining 1000 points.	Enemies spawn every 6 seconds with a score above 1000 but below 2000.	Pass

15	3	Destroy an enemy tank.	An explosion animation should appear over the destroyed tank and then disappear.	Pass
16	3	Destroy at least 5 Streaker enemies.	At least one silver coin should have appeared which the player tank can pick up.	Pass
17	3	Destroy at least 5 Striker enemies.	At least one gold coin should appear which can be picked up by the player.	Pass
18	3	Start a game and observe a silver and gold coin on the ground.	A shimmer animation should play every few seconds.	Pass
19	3	Cause a coin to drop but do not pick up.	The coin should disappear after 8 seconds.	Pass
20	4	Start a multiplayer game and check the HUD.	Both player's health bars should be visible along the top.	Pass
21	4	Start a game and destroy a tank.	The players score should start at zero and increase by 100 points when a streaker has been destroyed.	Pass
22	4	Start a game and pick up a silver coin and gold coin at least once.	The coin count should start at zero and increase by 5 for every silver coin and 10 for every gold coin.	Pass
23	5	Start a game and let the player get hit 5 times by a shell.	The player should disappear once the health bar reaches zero and should no longer be targeted.	Pass
24	5	Start a game and destroy the player tank.	Health bar should gradually restore.	Pass
25	5	Start a game and destroy the player tank. Wait for the health to regenerate.	Player should appear above or below the base once the health has fully regenerated.	Pass
26	6	Lose the game by letting the player base health reach 0.	The game over screen appears and the final score is displayed.	Pass
27	6	Lose the game to let the game over screen appear. Select the "Retry" button.	The game map should clear, and a new game should begin. The score, health bars, coins and base health should reset to default values.	Pass
28	6	Lose a game to let the game over screen appear. Select the "Exit" button.	The game should return to the main menu screen.	Pass

Appendix B – Project Diary

The project's diary can be found at https://users.aber.ac.uk/glc3/mmp/blog/
Or a copy is included below:

Greg's Project Blog

07/02/22 7th Feb

Created a shortlist of possible game ideas. Some of these ideas included a Wii Play Tanks clone, Space Shooter game with local multiplayer and a tower defence type game.

08/02/22 8th Feb

Decided to make a tower defence type game but with certain differences. First, the players and AI enemies will be tanks. Objective is to shoot invading tanks before they reach base. I had a group supervisor meeting today where I discussed the idea with others in the class.

17/02/22 17th Feb

Worked on project outline and blog today. A week behind to due being ill and unable to work. Thankfully I was still able to think through my potential solution!

22/02/22 22nd Feb

Had a group meeting to share and discuss visual elements of our projects.

07/03/22 7th March

Did research on important aspects of the game. Made a prototype solution which successfully takes inputs from an xbox one controller and moves a temporary sprite around the screen. More work will need to be done to determine the angle of the gradient.

08/03/22 8th March

Had a meeting with Neal Snooke and talked about progress including the working program I had made.

09/03/22 9th March

Changed core parts of my working prototype from Monday. While investigating, I encountered a major issue which I did not expect to encounter. On MS windows, gamepads are connected via the xinput api which is proprietary and therefore not included with linux (debian). Many button mappings, and especially analogue inputs are mapped differently and therefore, grabbing controller values directly would not work unless I needed to only support a very specific system. I created a GamepadController class to act as an intermediate and grab the intended inputs based

on the system. This is a work-in-progress but will hopefully make the code much more robust. I also changed the "entity" system with Object Oriented design. Entity is the super class which contains and gameplay element such as a wall, player or bullet. Tank sprites made using pixel image editor: piskell.

10/03/22 10th March

Removed temporary code and moved functionality to final location in the Player class. Small bugs regarding compatibility with Microsoft Edge have been fixed in the GamepadController. Game has currently been tested on Firefox 97.0.2 (Windows), Firefox 91.6.0 LSR (Linux) and Microsoft Edge 99.0.1150.36 (Windows). Another bug has been fixed in the GamepadController which prevents crash when removing controller during game loop. Added a reticle (cursor) to show what the user is pointing at which is invarient to magnitude of stick. Implemented tank turret rotation far earlier than expected - used atan() and pi if x is negative.

11/03/22 11th March

Small fixes in the gamepad controller to support more browsers (Samsung Internet).

20/03/22 20th March

Added a second tank cursor to highlight currently selected block. Also loosely implemented feature to build walls on tile which is currently selected. Currently multiple walls can be built on one tile so this needs to be fixed later.

21/03/22 21st March

I had my mid-project demonstration where I was able to demonstrate the basic player movements and mechanics. Following on, for the rest of the day I worked on implementing collision detection, tanks firing shells, including hit detection and bounce physics so they reflect off walls. I also implemented a mini health bar which appears over hit objects (such as walls) and displays how much health that block has for 5 seconds before disappearing.
Shells are instantiated once the user presses the right trigger on the controller and act independently from other objects.

29/03/22 29th March

Created a path class which will handle the pathfinding for the enemies. Managed to implement a function which reads the current game data and generates a pathfinding map. Also implemented a function to calculate the euclidean distance between two tiles which will be used later for pathfinding.

30/03/22 30th March

Implemented a function to print the current status of the path finding grid for debugging.

31/03/22 31st March

Implemented the A* pathfinding algorithm into the path class.

01/04/22 1st April

Finished the path class which can now be used to navigate NPCs. Created the streaker enemy class which will not shoot, but simply move to the player's base and explode upon impact. When instantiated, the enemy begins moving towards the base but can be intercepted by walls.

09/04/22 9th April

Started developing the main menu. I have implemented icons at the top of the screen to show how many controllers are connected.

11/04/22 11th April

Revisited the gamepad controller class to add functionality I had previously not been able to. Up to this point I had only partially implemented a function to get button presses so that I could test the ability to place walls. However, now since I'm adding menu functionality, it is important that other buttons work so that the user can control menu selections. I also needed to go back to windows to get button mappings there so I could make the game get the correct buttons irrespective of operating system.

14/04/22 14th April

Worked on the gamepad controller class to add the final bit of code to the getButton function so now d-pad button presses can be detected on Linux. I had to make a special exception for these as if statements because on linux they are treated as an axis and not a button, on windows. I then made it possible to change the currently selected button in the main menu by using the d-pad on the controler. When the "A" button is pressed, the relevant function is called.
The more I have been working on the UI, the more I encounter potential design issues due to confusing implementation. I have only just realised that the way everything is implemented doesn't make full sense, but I can mention this in my report.

22/04/22 22nd April

Designed an explosion animation in Piskel, consisting of eight frames of animation. I also created a class for this explosion effect which can be instantiated when an enemy tank is destroyed or detonates at the base. I have also created a simple animation system so that the game can loop through all of the frames with custom intervals between each frame. All of the frame intervals are the same for the explosion animation but will be useful when animating the coins which will be next.
I first designed a gold coin, but then decided to also design a silver coin when I realised that these could be dropped by the two different enemy tanks. The coins also have a shiny animation which currently plays once every two seconds using the simple animation system I designed with

the explosion effect. I have yet to actually implement the coins into the game though they have their own class and mostly working code.

23/04/22 23rd April

Implemented code into the enemy tank classes so that there is a 50% chance that a coin is spawned on destruction. The coin disappears when the player collides with the coin but I need to update this later as the collisions are slightly off since there is only one point in the centre of the coin being checked. I have also added another function to the coin class so that it despawns after a specified time.
I have also modified the tiles sprite so that walls show visual damage when health reaches 50% and 25%.
Today, I have also updated the player class script which deals with wall building so that walls cannot be built where they would block a path for the enemy to reach the base. I did this by creating a virtual copy of the gameData and adding a wall inside the copy. I then passed this virtual gameData to the pathfinder to find a path. If a path could be found then it is possible to put the wall there as it clearly doesn't obstruct the route that the enemy AI will take. Walls also cannot be placed on the map boundaries so that enemies can spawn there.

25/04/22 25th April

Today I added various new features and fixed a few bugs. The first major feature I implemented was writing the gamescreen class so that code which manages the overall gameplay can run. The first feature I implemented was the HUD (heads up display) which shows the player(s) health as a health bar at the top of the screen. Below that is other information such as the health of the base, as a percentage, the players score, and the amount of coins they have.

Today, I also added some important gameplay features to the tanks as they now process their health. This means that they can now detect what level their health is, and if it reaches zero, they are killed. When they are destroyed, they are moved offscreen out of sight and their update cycle is mostly stopped (only to check if they are still dead). While in a dead state, their health will slowly recharge until it is back to maximum. At this point, the tank will be revived and will be moved back into the game (revived).
Today, I also fixed a known bug where if a wall was built in an enemies path after it had already spawned, it would hit the new wall and stay there. This was due to the fact that the enemies path would never be re-checked, even after a collision. However, now another path is immediately calculated as soon as the enemy hits a wall.
Another fixed bug involved a situation where striker enemies would always shoot at player 1, if they were in target range, even if player 2 was closer. This was due to the target select loop immediately exiting once a target had been found, while ignoring other potential closer targets. The fix was to add a variable to store the shortest distance before updating the target entity.

26/04/22 26th April

Today's main goal was to get the enemy spawner working, and that's exactly what I have done. Both types of enemies now spawn from each corner of the screen after the game starts. Game

difficulty data is stored in an array and the difficulty is selected based on the player's current score. As the score goes up, the delay between each spawn decreases and the chance of a striker spawning increases.
In order to deal with multiplayer balance, this has been solved simply by spawning two enemies instead of one each cycle when there are two players.
An additional "help" button has been added to the main menu too, and the game over buttons have been added which appear when the base's health reaches zero.
A bug was found where enemies would often break when they have been encircled with walls and cause large performance drop as they would keep trying to find a new path. A simple solution is actually to have them destroy themselves in such situations as they have not been designed to break out themselves and it is not possible to add that functionality this late. However, this is not much of an issue since this is an unusual and not very effectively strategy anyway. Enemies destroy themselves if they cannot find a path to the base.