

Video Game: Dead Man's Volley

CS39440 Major Project Report

Author: Peter Hodgkinson (peh19@aber.ac.uk)

Supervisor: Dr David Hunter (dah56@aber.ac.uk)

28th April 2019

Version 3.2 (Release)


This report is submitted as partial fulfilment of a BSc degree in
Computer Graphics, Vision And Games (G450)

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 

Date 28/04/2019

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 

Date 28/04/2019

Acknowledgements

I'm grateful to the test users that play tested the game and gave honest and constructive feedback.

I would like to thank David Hunter (dah56@aber.ac.uk) for being my supervisor and Alexandros Giagkos (alg25@aber.ac.uk) for being my second marker.

Abstract

The purpose of this project is to produce a game that focuses on the common gameplay trope known as 'dead man's volley' or the 'tennis boss'. This game was made in Unity because its use of behaviour scripts and versatility could prove a challenge with no prior experience using the editor.

The final version as presented for this project has the player deflecting projectiles back at the enemies that fired them to score points. This is done in an arena like level with scripts to manage the gameplay. As expected, learning to use Unity did provide a challenge, with minor hiccups during development, however, ingenuity and clever application of the resources provided by the game engine helped resolve most of these issues.

Users who tested the game said they enjoyed playing it and as the reason for making a game is so that people can enjoy it, this project can be considered a success, even though the software is incomplete.

Contents

1. BACKGROUND, ANALYSIS & PROCESS.....	6
1.1. Background	6
1.2. Analysis	6
1.3. Process	7
2. DESIGN	8
2.1. Overall Architecture.....	8
2.2. Detailed Design	8
2.2.1. Player	8
2.2.2. Enemy.....	10
2.2.3. Managers	12
2.3. User Interface Design.....	13
3. IMPLEMENTATION.....	16
4. TESTING	17
4.1. Overall Approach to Testing	17
4.2. Automated Testing	17
4.2.1. Unit Tests	17
4.2.2. User Interface Testing.....	18
4.2.3. Stress Testing	18
4.3. User Testing	18
5. CRITICAL EVALUATION.....	18
6. ANNOTATED BIBLIOGRAPHY.....	20
7. APPENDICES.....	21
A. Third-Party Code and Libraries	21
B. Ethics Submission.....	21
C. Code Samples.....	23
D. Additional Diagrams.....	29
E. Test Table	30

1. BACKGROUND, ANALYSIS & PROCESS

1.1. Background

In preparation for this project, initial research was done on several common game mechanic and tropes that could be further explored. It was decided the project would focus on one called 'dead man's volley', also known as the 'tennis boss' [1]. This mechanic is first recorded to appear on the game 'The Legend of Zelda: a link to the past' [2] and has appeared in many franchises since then. The mechanic consists of the player and an enemy parrying a projectile towards each other until one is hit by the projectile in an area of the body that can take damage. It is most often performed during a boss fight with special projectiles or a special move that the player can perform in combat.

With the primary concept of the game decided, further preparation could be taken. Simple 3-d models were made for the player, several types of enemies and a projectile. These models were given several animations for the planned functions in this game. Their simple designs were to reduce the amount of time spent working on them and were inspired by the game 'Thomas was alone' as that game uses simple geometric shapes for the characters and has a narrator build the complex and endearing story.

The last bit of preparation was to brainstorm what kind of gameplay should be developed for the project. While an online vs mode would show off the most skill in implementation, it would take too much time and effort to program within the project timeframe. The initial plan was to create a set of explorable levels with a loose story for the user to explore, however, if time became an issue, an arena combat mode could be implemented a lot quicker.

The motivation for this project is to simply enjoy designing and producing a game with the intent of learning new skills and developing current one tied to game development.

1.2. Analysis

As the problem was to produce a game, and the plan was to design it to focus on the mechanic 'dead man's volley', the next step was to further analyse the problem and the plan to develop a list of objectives and a primary goal. The first thing to decide was what game engine the program would be developed in, as different game engines and editors may affect how much of the project could be done on time and to what quality.

After a bit of debating it, the plan was to make the game in Unity [3]. While Unreal is capable of great visual aesthetics and its blueprint visual scripting is easy to understand, Unity is much more versatile with its use of behaviour scripts and would be much easier for the manager to examine. With more experience using Unreal and none using Unity, this was an interesting challenge to learn how to program in the unity editor using C#.

The next part was to identify the key features that would need to be developed to make the game. These were the primary game objects that the gameplay would be focused around and needed to be the most developed. The key features of this game were: -

- A player game object that the user can control. This player can move and rotate to face a direction independent from the direction it is moving. The two actions this player can do are to swing a bat to redirect projectiles in front of it or bring the bat up into a guard state to deflect the projectiles. The player can control the direction of the projectiles hit by the bat

swing by pointing the player in the desired direction but have little control of them if they use the guard as that works similar to light bouncing off of a mirror.

- Enemies that attack the player by shooting projectiles at them. These enemies only start shooting the player when they are within a given range. By default, these enemy's move towards the player, when they are within a set range, they will start to circle the player to prevent them from being easy targets. If the player gets too close, the enemy will attempt to move away by heading backwards.
- Projectiles that have different effects depending on what they collide with. If they collide with the player or an enemy, they will deal a given amount of damage. The effect of hitting a wall can be that the wall or projectile is respawned, or the projectile simply bounces off dependent on the type of wall the projectile hits. The projectile should also have states so that the player can't be damaged by it should the player have just redirected it or the enemy if they have just fired it. On collision with any wall, the projectile should be set to a neutral state that can damage either the player or enemies.

If time had not been an issue, ideally all brainstormed gameplay ideas could have been implemented. Other things that could be added include a functioning options menu to allow the user to adjust the games setting to their personal preference, and a 'vault' where users can preview the models, notes from planning, the music and sound effects.

With the amount of time available being limited and some being put aside to learn to operate Unity, it was initially determined that a few levels of a story mode could be made within the time available. However, in case it took longer than expected to implement the key game features, a single arena survival level would suffice.

1.3. Process

Each asset and behaviour were developed using a feature driven development methodology, focusing on the features that make up each asset. Development of the game as a whole followed a waterfall development style as many assets required something else to be developed beforehand i.e. before projectiles can do damage, player health needs to be implemented or before an enemy spawned can be made, at least one enemy type must exist.

2. DESIGN

2.1. Overall Architecture

The key parts of software architecture that needed to be considered for this project were: -

- To keep each class as simple and modular as possible to reduce repeated code and avoid systems being directly dependent on each other.
- The code should be easy to adapt and develop so that future developers that pick up the code can easily understand what is going on. This also makes it easier to further develop the game beyond the scope of the project should that become a desire.
- Each script should be easy to debug and remove errors. This part should be supported by the first two key parts.

Unity helps with achieving these architecture requirements through the use of its built-in class called `MonoBehaviour`. All script written in Unity use the `MonoBehaviour` class as a parent class which provides a slew of methods that most game object uses, such as `Update()` which is called every frame, or `Awake()` which is called when that instance of the script is first being loaded.

2.2. Detailed Design

2.2.1. Player

The 'player' is the primary game object that the user controls. It must feel responsive to input to prevent the user from having difficulty controlling it.

- Movement

Due to the nature of the design of this game, movement can be broken down into two key parts; translation of the object and rotation.

Translation is performed by creating a vector from horizontal and vertical axis inputs. This vector is then multiplied by the time between frames and a value for the player's movement speed. The vector is then applied using the rigidbody function, `MovePosition()`, that translates the object using the vector as coordinates to travel to. The reason for using the function `MovePosition()` instead of `position` is that the former renders intermediate positions between the former and target coordinates creating a smoother transition; whereas the latter does not, making it look more akin to teleportation.

Rotation is done by performing a ray cast from the camera at the mouse's position and getting where on the floor it hit. If the ray successfully hits the floor, a vector is created of where the ray hit to the player's location. This vector is then used to produce a Quaternion, a variable used to represent rotations. This converts the vector into a rotation that points from the player to the mouse, all that is left to do is apply the quaternion to the player's rigidbody using the `MoveRotation()` function. `MoveRotation()` is used instead of `rotation` for the same reason as `MovePosition()`, intermediate rotations are rendered to display a smoother transition.

• Health

In this game, the level is over if the player loses all of their health. The amount of health is a numerical value, for simplicity, an integer seems to be the best option. Health is removed using a public function that the object dealing the damage will call. The function removes

that amount of health required, then alters the player model to visually identify the amount of health the player currently has left. The lifeline diagram below (Figure 1) displays this function when being called when taking damage from a projectile. If the player has lost all of their health, they are considered dead, so all functions tied to the player are disabled and the model is rendered invisible.

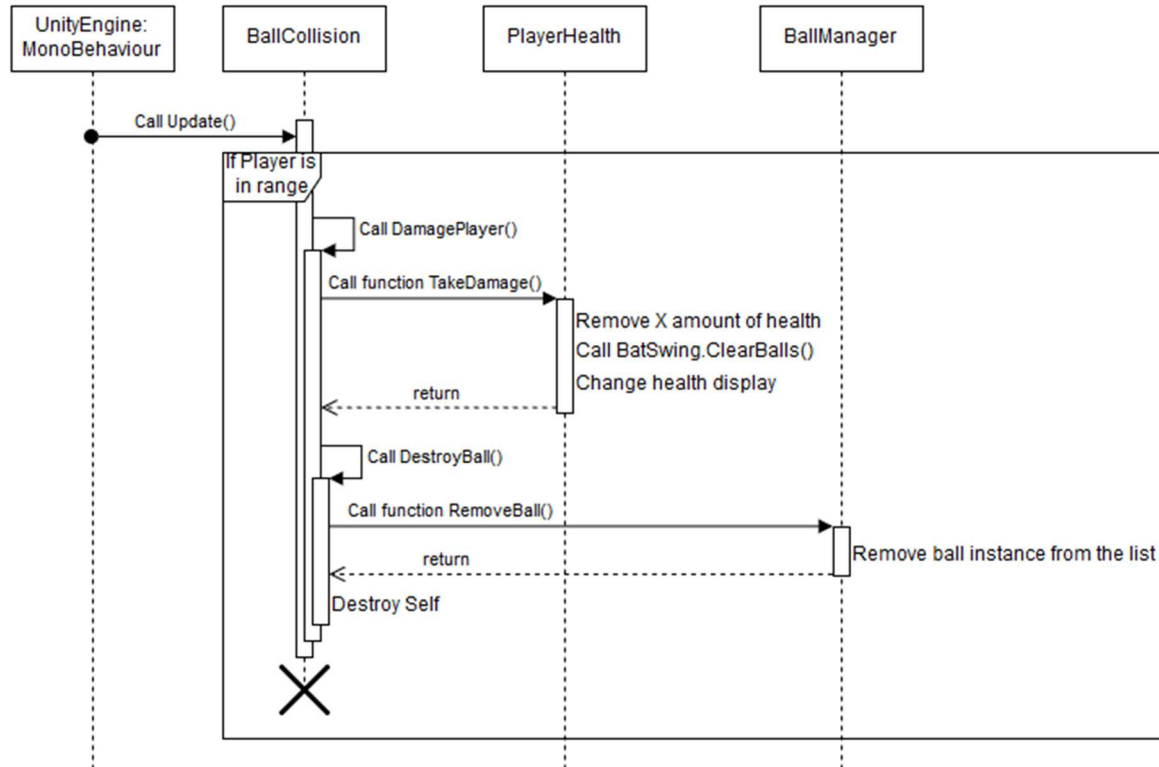


Figure 1. A lifeline diagram of a projectile damaging the player on collision.

- Actions

The player has two possible actions they can take. They can either swing a bat to redirect a projectile towards the players forward direction or they can use the bat to guard to protect themselves from the projectile with it bouncing away similar to light bouncing off of a mirror.

To set the 'swing' action, an empty will be attached to the player with its own set of trigger colliders that will act as the bat swing; any projectile in that area when the action is performed will be affected. When a projectile enters this area, it is added to a list so that all projectiles in the area can be affected at once, if it leaves the area it is removed from the list. To apply the effects of swinging the bat on all the projectiles in the area, a for each loop is used on the list where the velocity of the projectile is changed to a new projectile speed multiplied by the forward direction of the player. The state of the projectile is then changed so that it cannot damage the player unless it collides with something else first. The lifeline diagram below (Figure 2) displays this function in action.

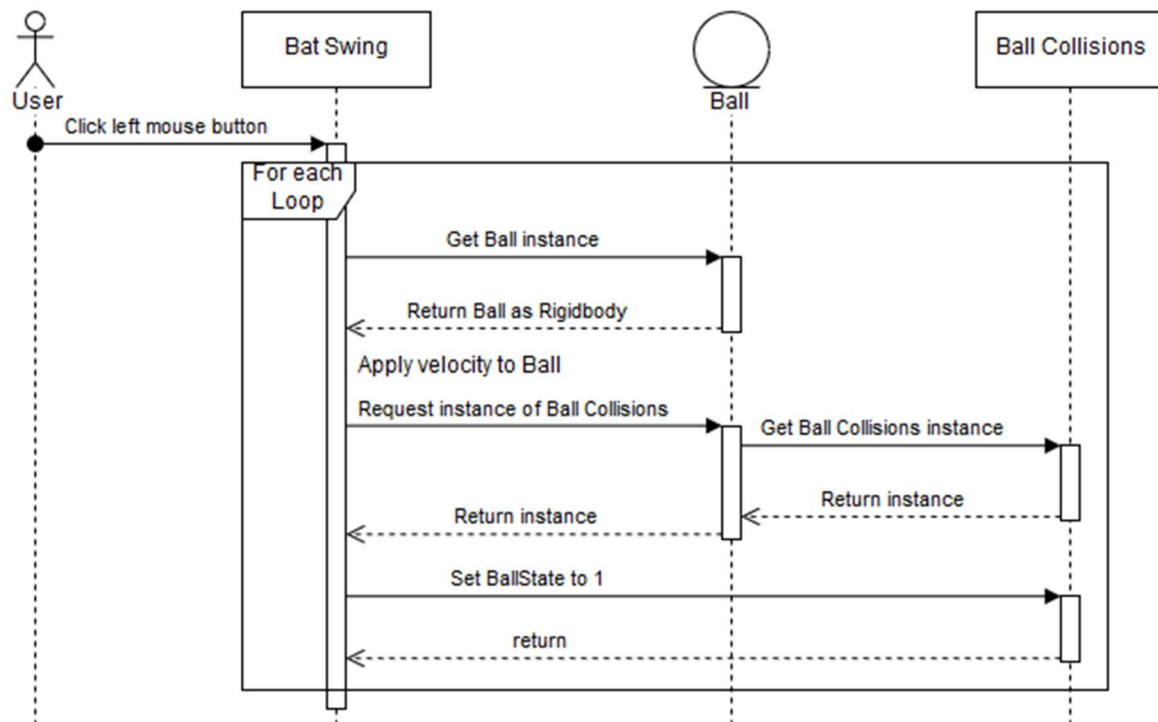


Figure 2: A lifeline diagram of the bat being swung to redirect projectiles.

To create the 'block' action, an empty will be attached to the player with a collider that will act as the bat guard. When the scene is started, the collider is disabled as the player doesn't start the game blocking. A Boolean variable called guard state is also set to false as this is how the collider will be controlled. Using the animator that controls the player's animations as a reference, if the guard state Boolean is not the same value as a Boolean in the animator called `IsGuard`, the colliders current state as to if it is enabled or not is switched and the guard state is set to the same value as `IsGuard`. With the collider enabled in front of the player, it acts as a wall that follows the player's movements. To discourage solely relying on this action, a function in the movement behaviour changes the movement speed of the player to a lower value whenever the `IsGuard` Boolean from the animator is detected as true. The movement speed is reset to its original value if the `IsGuard` Boolean is detected as false.

2.2.2. Enemy

Enemies are the primary obstacle the user will have to overcome. For the version of this game that will be produced for the project, only one type of enemy will be implemented, however, the addition of more enemy types with slightly different behaviours should be considered so code should be written with this in mind, either future proofing it now or making it easier to adapt later.

- Movement

Enemies are to always face the player; this makes aiming projectiles simpler. To achieve this, a similar method to the player's rotation is used. In this case, a vector from the enemy's position to the player's position is calculated. This vector is normalized, then applied to the enemy using a Quaternion function called `Slerp()`, which rotates an object from one vector direction to another over a period of time.

To move the enemies around the environment, a class that is provided by Unity called NavMesh will be used. NavMeshes are used to perform spatial queries, most often related to pathfinding. To set up the NavMesh, all objects that are used as the environment are to be set as 'navigation static'. A component called a NavMesh Agent also has to be attached to the enemy, which details about the agent's obstacle avoidance, pathfinding and steering. To make the agent move, a function that is part of the nav mesh agent called SetDestination() is used. This function takes in a vector3 to set as the destination point for the agent.

The enemies should appear to be capable of some sensible tactics, such as maintaining a distance to shoot at the player or retreating should the player get too close. To detect how close the player is, two empties with sphere colliders are used, with one collider larger than the other. These colliders are used to detect the player by passing a Boolean to the movement behaviour if the player enters them. If the player is in contact with the smaller collider, the destination of the NavMesh agent is set to behind the agent by removing the forward of the enemy from its position saving it as a vector3. This vector is then set as the nav mesh agent's destination. On the other hand, if the player is in contact with the larger collider, an additional Boolean is calculated. This Boolean is calculated so that the more frequently it is one value, the more likely that it will be the other value on the next update. This Boolean is to decide if the enemy will move to the left or the right. Similar to moving backwards, this is done by setting the destination to a vector of the agent's position plus or minus the enemy's right direction. If the player is not in contact with either of the colliders, then the destination is set to the player's current position, however, if either the player's or the enemy's health is reduced to 0, the NavMesh agent is disabled and the enemy stops moving.

- Health

The health of the enemy can be controlled in much the same way as the player's health, as an integer that is reduced in value when the enemy collides with something then is meant to deal damage. This behaviour is the easiest to future proof as similar to the player, the enemy health is displayed as part of the model. By extending the statement that deals with the visual display of the unit's health, enemies can have a wider variety in the amount of health they have. This will not be used to its fullest extent in the version provided as part of the project as it will only have one type of enemy.

- Shooting

For the enemy type that will be produced for this project, its main action is going to be producing a projectile and giving it a velocity towards the player. To make the enemies appear more intelligent, they should only shoot if the player is within a range. To implement this, an empty with a sphere collider will be attached. If the player overlaps this collider, a Boolean value is passed on to the main behaviour.

To shoot the projectiles, a function of MonoBehaviour called InvokeRepeating() is accessed. This function calls a method by its name repeatedly with a defined amount of time between calls. To prevent all the enemies spawned into the environment from all shooting at the same time, a random number is selected from a range for the repeat time of the method for each enemy. The method for spawning projectiles gets an instance of a new projectile from the projectile manager and sets its initial position to an empty in front of the enemy. This projectile is then given a velocity of the empty's forward multiplied by a projectile speed and set to a state so that it will not harm other enemies until it has collided with something else first. This function is shown in the lifeline diagram below (Figure 3).

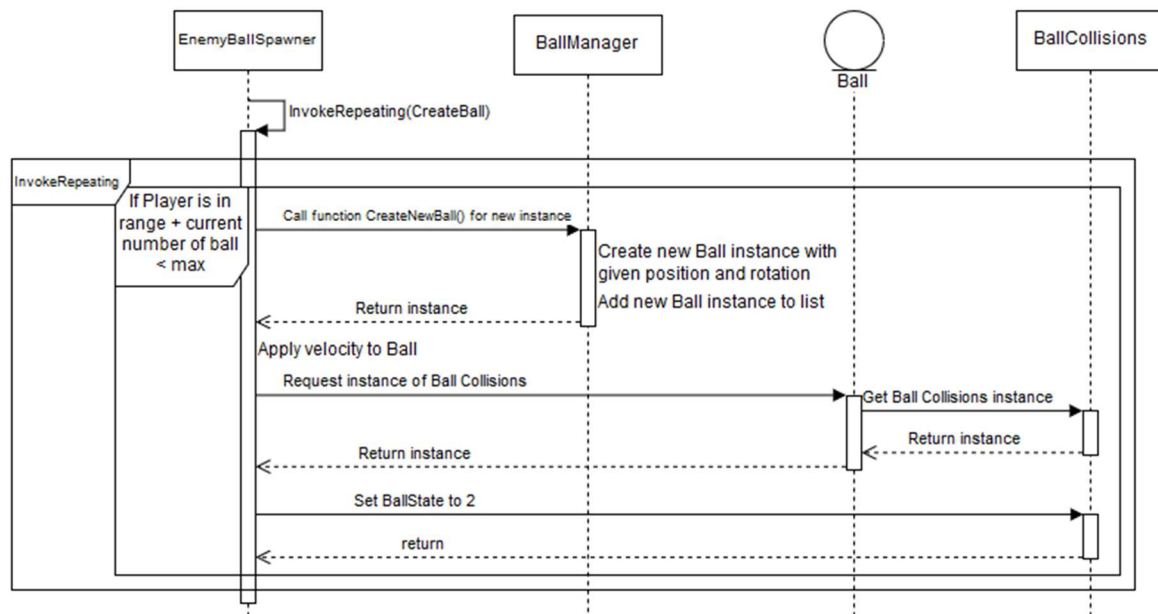


Figure 3: A lifeline diagram of an enemy spawning a projectile.

2.2.3. Managers

To control the different scenes, several behaviours titled as managers will be implemented by attaching them to a single empty game object in the environment.

- Enemies

To control the spawn rate of enemies, two behaviours are used. The first is called the 'Enemy Spawner Manager'. This behaviour acquires a list of game objects that will be used as enemy spawn points, then calls the function `InvokeRepeating()` to start spawning the enemies. The method used starts by generating a random number from a range based on the number of game objects in the list. This number is then used to get an instance of the game objects behaviour that spawns the enemies so that a public function of that behaviour can be called to create the enemy. This function calls on the second manager behaviour related to enemy spawn to create a new instance of an enemy. The second behaviour is the 'enemy manager' which is used to create new instances of enemies and store references to them in a list. New instances will only be created if the current number of stored instances in the list is less than a defined value for the maximum number of enemies that can be spawned in at once. This is to control the number of enemies to reduce the risk of the frame rate reducing due to the number of game objects. When an enemy has its health reduced to zero, they are removed from the list before being destroyed to allow more enemies to be spawned. The process of spawning a new enemy is shown in the lifeline diagram below (Figure 4).

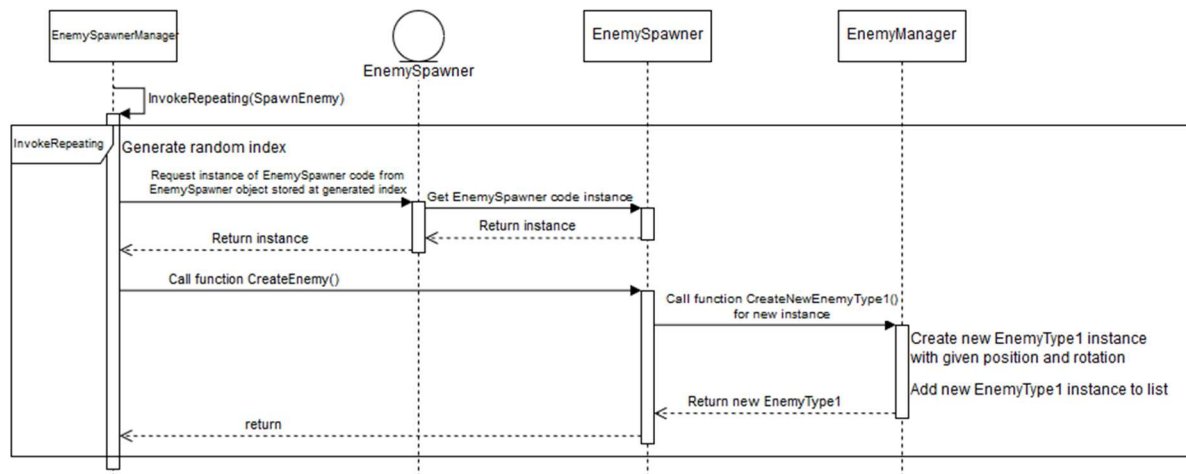


Figure 4: A lifeline diagram of an enemy being spawned

- Projectiles

To control the number of projectiles in the scene at once, another behaviour is used that is functionally the same as the enemy manager called Ball manager. This behaviour implements a list to store instances of projectiles so that the current number of projectiles in the scene can be determined with ease. It also holds the public method that is used to spawn in projectiles at a given position and well as the method to remove instances of these projectiles from the list when destroyed.

- Menus

The behaviours used to control the menus are mostly simple. The 'Main Menu Manager' is used to control the projectile spawners in the main menu scene as they are used for decoration. This behaviour is a near one to one copy of the 'Enemy Spawner Manager'. The 'Start Menu Manager' behaviour is used to transition from the title screen to the main menu. The 'Pause Game Manager' will either display a small menu and set the time scale of the scene to zero to pause time or hide the menu and set the time scale to one dependent on a Boolean variable that is changed every time the pause button is pressed. The 'Game Over manager' uses an instance of the player's health to determine when the user has lost. If the player's health is reduced to zero, the time scale is set to zero, the game over menu is enabled and an animator is used to display both the menu and a game over message.

2.3. User Interface Design

While there was no intention on implementing a scoring system into the game as it would encourage the user to try and get a high score instead of surviving, a simple one was implemented into the version to be produced for the project to provide a goal for users. This is displayed during gameplay as an integer in the top left corner of the screen and increments every time an enemy is destroyed.

The players health is displayed as part of the model, shown as coloured segments. When the players health is reduced by a determined amount, a segment is disabled so that it is no longer rendered.

Menus are displayed in a vertical format to make the desired option easier to identify and select. The buttons are designed to mimic the design of the player and enemy models to be consistent with the visual style of the game. A behaviour is implemented so that the user can select a menu option using either the keyboard or the mouse. Behaviours are also used to load scenes and end the application. The behaviour for changing scenes has two method, one that sets an integer for the index of the scene the user wants to transition to, and one that loads that scene. The behaviour for

ending that application has one method that accesses the application run-time data to quit the player application. Below are some flow diagrams (Figure 5, Figure 6 and Figure 7) of the main menus and their current functionality.

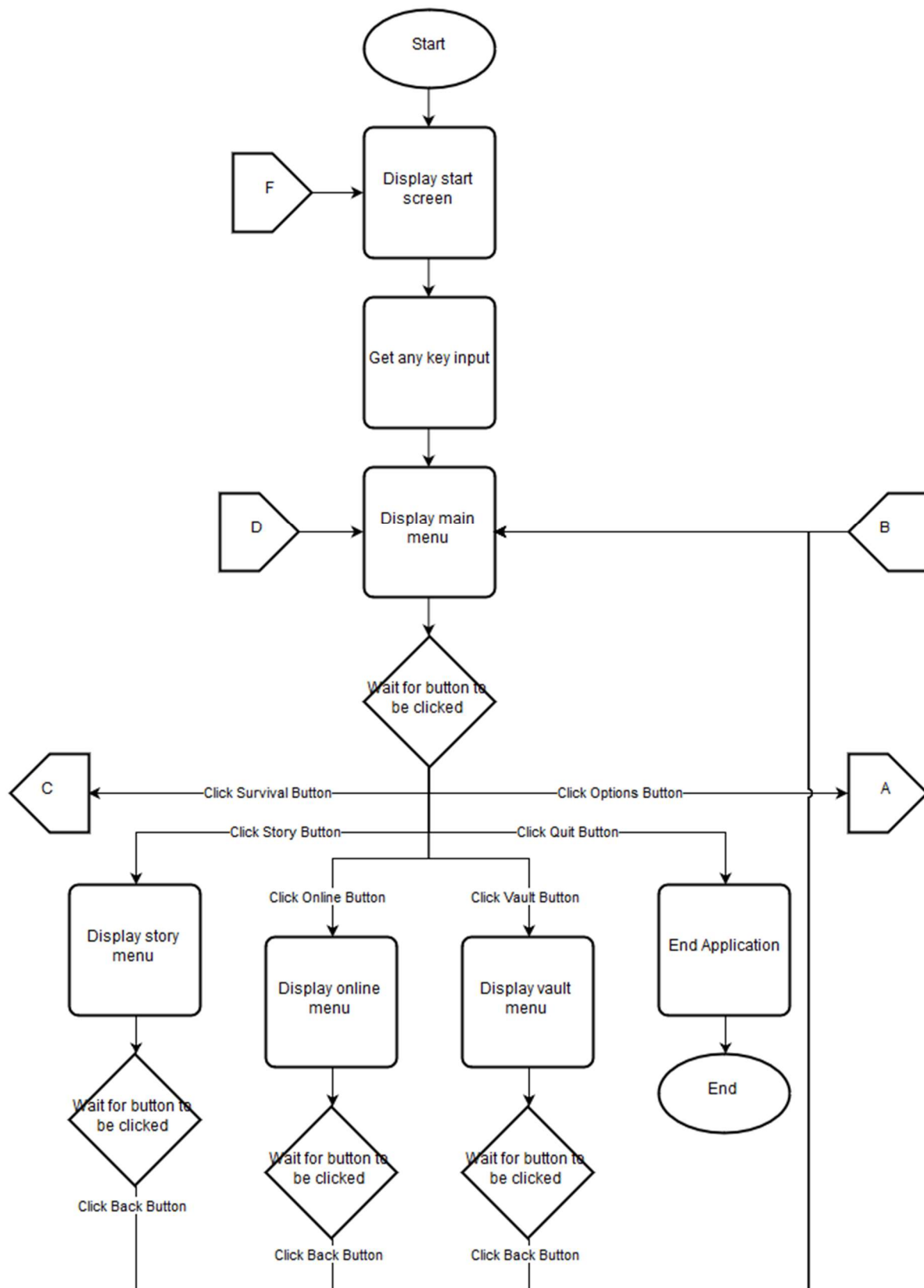


Figure 5: A flow diagram of the start menu and most of its functions.

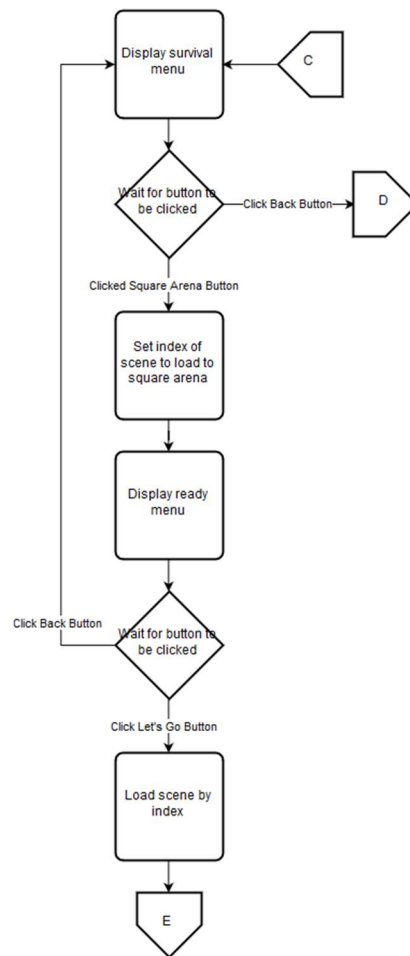


Figure 6: A flow diagram of the level select menu currently implemented.

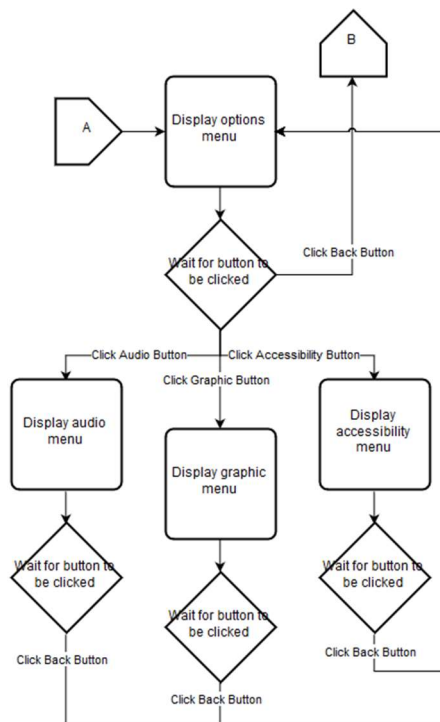


Figure 7: A flow diagram of the options menu currently implemented.

3. IMPLEMENTATION

Some issues did occur during implementation, most were due to a lack of knowledge of methods that Unity provides or documentation on such method being sparse or poorly communicated. The primary example of this issue occurred during implementation of the player's animation. As the player was to display an animation then enter a new idle state, it was thought a code-based solution was required. After a week of researching who to solve this problem, a method that required a single checkbox to be set to true was brought to light. This meant that much more time than what was required was spent on this issue.

The next issue to occur was the initial implementation of the behaviour for swinging the bat. The first version of this behaviour only affected the last projectile to enter the collider. The solution to this was to store a list of instances of the projectiles rather than a single variable but did require a large portion of the behaviour to be rewritten to compensate for this design change. This took a day to plan and implement these changes.

After this, the next issue was that some of the game objects were difficult to distinguish from the background. While in theory, an easy solution would be to change the colours to be brighter, testing on multiple computers showed that the colours were inconsistent as sometimes the floor would be a different colour to the background of the sprite that displays the controls. The colour difference is a minor issue that doesn't affect the gameplay and so will not be addressed in this version of the project.

A reoccurring issue was poor grammar or syntax in the code due to it being written quickly, resulting in either function not working properly or the editor refusing to test run a behaviour. While these occurrences were easy to deal with, they were frequent enough to lose a substantial amount of time making the required corrections.

The libraries provided by Unity greatly assisted with the implementation of the code. They provided methods and functions that would have taken time to develop and fine tune, resulting in not having enough time to develop an actual game. The only issue was that the documentation of the methods and functions was not concise or easy to find in some cases, resulting in time lost researching for particular methods, that would simplify implementation.

Some behaviours were much simpler or quicker to implement than expected. This freed up time to refine and optimise code or develop methods to optimise the gameplay by preventing any loss of framerate. This led to the development of the ball and enemy managers that were talked about previously.

Comparing how the key features of the game turned out in comparison to their initial planned requirements: -

- The player's movement feels quick and responsive, working well with the rotation which feels smooth intuitive. Swinging the bat feels great as it is instant, however, it does feel like it doesn't register a projectile it should have hit every so often. This could be due to the size of the bat area, which can be adjusted. The guard feels sturdy but sluggish; it may be that drastically reducing the player's movement speed when guarding was a poor way to discourage overuse of the action. An issue that has arisen is that rarely, but sometimes a projectile tunnels through the guard and hits the player. This occurs rarely enough that the issue can be left alone for now, but as development is planned to continue after the project, it will be a high priority problem to solve. Overall the player feels really good to control but would definitely benefit from further tweaking.

- Enemies provide a substantial challenge to the user. They seem relatively intelligent with their movement changing dependent on their distance from the player as intended. The movement isn't as refined as intended as the enemies seem a bit slow when moving backwards and stutter a bit before circling the player as they decide which direction to circle in. Ideas on how to smooth out these issues are being considered but won't be implemented until after the project. The enemy shooting is somewhat varied, not all the enemies shoot at the same time, but a lot do so. Increasing the range for the start and repeating of the shooting method timer would likely help give more variety. The enemies can be quite difficult to hit once they start circling the player and can quickly overwhelm once more than four are within range. The obvious solution is to decrease the enemy limit and spawn rate to reduce the risk of being overwhelmed. Overall the enemies have turned out really good for the amount of time put into their development.
- The projectiles turned out quite good for their simplicity. Their initial velocity isn't too fast that the user can't keep track of them, but not too slow that they don't pose a threat. The number of projectiles that can be in a scene can get overwhelming sometimes but that can be adjusted in future development. An interesting issue that was brought to attention was that projectiles sometimes lose their velocity and stop moving, this results in them never being in a situation where they despawn. A solution that has been considered has the ball delete itself if it is not in view of the camera and its velocity has a magnitude, however, this will be something that will be developed after the project.

4. TESTING

4.1. Overall Approach to Testing

To ensure that each function worked properly, extensive testing was required. After a method had been written, its effects on the scene would be tested a minimum of ten times in a rudimentary scene created for the purpose of testing all the key features. This scene had additional methods created to streamline testing by allowing projectiles and enemy's to be spawned using keyboard input as well as their normal timer functions. A method that reset the scene was also used so that time wasn't wasted constantly stopping and starting the testing environment.

The reason for testing each function a number of times was to ensure that no random element would prevent the function from working properly. Once the game object the methods were being attached to was presumed finished, it was tested another ten times to ensure that all the methods would work well together, and nothing would clash.

Once all the key game object had been developed, any further testing would be done in the scene the object would be implemented in. Primarily this was due to the functions being tested being quite simple, though in some cases it was due to that function being exclusive to that scene.

4.2. Automated Testing

4.2.1. Unit Tests

Unit testing was not used during the testing process of the game. This was primarily due to most of the functions having a visual output in the scene that can be tested and observed

for. An example of this would be the ball manager as if there are never more the set maximum number of projectiles in the scene at any one time, the function works.

4.2.2. User Interface Testing

Interface elements were tested just as much as the game objects. Most of these elements have simpler behaviours than the game objects and so were easier to test. These user interface elements are not the primary focus of this project, so while they were tested thoroughly, debugging them was not a high priority.

4.2.3. Stress Testing

Stress testing was done by spawning an increasingly large number of enemies and projectiles until the frame rate of the program started to decrease. This wasn't an optimum test as the computer systems it was being tested on were more powerful than the average desktop and didn't show any signs of the frame rate dropping. Behaviours were still implemented to limit the amount of spawned object as it is likely that some computer systems will not be able to handle the same amount of game object as the systems we tested on.

4.3. User Testing

User testing was conducted on a small group to determine if the game was enjoyable to play. Test participants were given two different versions of the game to try. The first version was of the testing environment once all the key game object had been developed. This was to get their input on the visual aesthetics of the environments and how it felt to play this demo. The second version presented was the one that will be the final version to be handed in for the project. This test was to determine the appeal of the menus and the amount of fun the user got out of playing the game. No formal documentation was made on the result of these tests as it was primarily to make sure that users would enjoy playing this game, however, constructive criticisms were noted down to be considered for future development.

5. CRITICAL EVALUATION

To evaluate how well the software turned out, the test users said that what they played was a success as they enjoyed playing the version of the game that they were given. I, however, would say it turned out ok, as there were a good number of features that I didn't have time to implement and a few chunks for code that could have been better future proofed or optimised.

I'd say that my identification of what was a requirement for this game to work was correct, though if time had not been as big a factor, I would have added more to the list such as power-ups for the player or more enemy types. This is echoed in comments from some of the test users as one suggested a power-up that allows the player to recover health. While this seemed like a great idea, I simply didn't have the time to implement it.

One of the objectives of this project was to learn how to use the Unity Editor. I believe that this is successful as I'm quite confident in my skills in using Unity at this current point, more so than my skills in Unreal which I had quite a bit of experience in using beforehand.

Appearance and design wise, I'm really happy with how everything turned out. While the images used for the buttons and the controls aren't particularly polished, they give a nice charm to the game. The 3-D models really do compliment the gameplay with their simplicity though I do wish I had implemented a way for the player to be able to choose from a selection of modes to use to add an element of customisation.

If I were to start this project again from scratch, assuming that all knowledge of how to use Unity was wiped from my mind to make it a fair comparison, I would most likely put more effort in to properly managing my time. While time management is a generic and frequent thing people say when asked what they would do differently, it's still true as I don't think I managed my time working on this project as effectively as I could have. I did have other assignments to get on with and I did need to take breaks from working to make sure I didn't lose my mind from just working nonstop, but in hindsight, those breaks didn't need to be as long or as frequent as they were.

Another thing I would do differently would be that I would start with the arena battle idea that you see in the current version. This would give me more time to properly plan it out, maybe have more than one model for the arena, maybe have a mode for score and a mode for timed survived. I would also try to implement a scoreboard for each level and mode, mainly just because that seems standard now.

I do plan to continue developing this game in the future as it was a lot of fun to work on and it could make for a good demonstration of skill for a job application. I won't be able to sell this game anywhere like Steam due to the intellectual property rights as mentioned in the document, MMP_S01 Information For Students, however, I'm more focused on providing an enjoyable experience to the players. Given that though, I might have gone with a different gameplay trope if I had to start again, as this seems like an idea that could be quite profitable if further developed.

Overall, I had a lot of fun working on this project. The game turned out all right by my standards for games and I think I have done a rather good job of explaining it in this report. The only way I can explain it any better is for whoever is reading this report to go and play the game. If you do, I hope you enjoy it.

6. ANNOTATED BIBLIOGRAPHY

- [1] TV Tropes. Tennis Boss, 24 August 2010. Available at: <https://tvtropes.org/pmwiki/pmwiki.php/Main/TennisBoss>. [Accessed 28 April 2019]
- [2] Zelda Wiki. The Legend of Zelda: A Link to the Past. 7 January 2013. Available at: https://zelda.gamepedia.com/The_Legend_of_Zelda:_A_Link_to_the_Past. [Accessed 28 April 2019]
- [3] Unity. Unity. 14 Mar 2019. Available at: <https://unity.com/>. [Accessed 28 April 2019]
- [4] Unity. Survival Shooter tutorial. 3 July 2015. Available at: <https://unity3d.com/learn/tutorials/s/survival-shooter-tutorial>. [Accessed 28 April 2019]
- [5] Unity. Tanks tutorial. 14 October 2015. Available at: <https://unity3d.com/learn/tutorials/s/tanks-tutorial>. [Accessed 28 April 2019]
- [6] Unity. Creating A Main Menu. 8 September 2016. Available at: <https://unity3d.com/learn/tutorials/topics/user-interface-ui/creating-main-menu?playlist=17111>. [Accessed 28 April 2019]
- [7] Brackeys. PAUSE MENU in Unity. 20 December 2017. Available at: <https://www.youtube.com/watch?v=JivuXdrIHK0>. [Accessed 28 April 2019]
- [8] Newcastle University. Technical Requirements Checklists. 8 December 2011. Available at: <https://research.ncl.ac.uk/game/mastersdegree/workshops/technicalrequirementschecklists/Technical%20Requirements%20Checklist%20Workshop.pdf>. [Accessed 28 April 2019]

7. APPENDICES

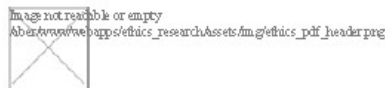
A. Third-Party Code and Libraries

Unity Engine – The project was built using the Unity Engine [3], a game engine that handles the graphics, audio and physics of the game.

Tutorials - Unity provides several sets of tutorials to assist developers with learning how to use the Unity editor. For this project, code from the 'Survival Shooter' tutorial [4], the 'Tanks' tutorial [5] and the 'User Interface' tutorial [6] was utilised, some of it was modified for use.

Another tutorial that had been published by Brackeys [7] was also utilised. The code from this one was used without modification.

B. Ethics Submission



22/02/2019

For your information, please find below a copy of your recently completed online ethics assessment

Next steps

Please refer to the email accompanying this attachment for details on the correct ethical approval route for this project. You should also review the content below for any ethical issues which have been flagged for your attention

Staff research - if you have completed this assessment for a grant application, you are not required to obtain approval until you have received confirmation that the grant has been awarded.

Please remember that collection must not commence until approval has been confirmed.

In case of any further queries, please visit www.aber.ac.uk/ethics or contact ethics@aber.ac.uk quoting reference number **12232**.

Assesment Details

AU Status

Undergraduate or PG Taught

Your aber.ac.uk email address

peh19@aber.ac.uk

Full Name

Peter Hodgkinson

Please enter the name of the person responsible for reviewing your assessment.

Reyer Zwiggelaar

Please enter the aber.ac.uk email address of the person responsible for reviewing your assessment

rrz@aber.ac.uk

Supervisor or Institute Director of Research Department

CS

Module code (Only enter if you have been asked to do so)**Proposed Study Title**

Major Project: Video Game: Dead Man's Volley

Proposed Start Date

22/02/2019

Proposed Completion Date

03/05/2019

Are you conducting a quantitative or qualitative research project?

Qualitative

Does your research require external ethical approval under the Health Research Authority?

No

Does your research involve animals?

No

Are you completing this form for your own research?

Yes

Does your research involve human participants?

Yes

Institute

IMPACS

Please provide a brief summary of your project (150 word max)

Produce a computer game that explores and expands on a common gameplay trope known as the tennies boss, or dead man's volley.

I can confirm that the study does not involve vulnerable participants including participants under the age of 18, those with learning/communication or associated difficulties or those that are otherwise unable to provide informed consent?

Yes

I can confirm that the participants will not be asked to take part in the study without their consent or knowledge at the time and participants will be fully informed of the purpose of the research (including what data will be gathered and how it shall be used during and after the study). Participants will also be given time to consider whether they wish to take part in the study and be given the right to withdraw at any given time.

Yes

I can confirm that there is no risk that the nature of the research topic might lead to disclosures from the participant concerning their own involvement in illegal activities or other activities that represent a risk to themselves or others (e.g. sexual activity, drug use or professional misconduct).

Yes

I can confirm that the study will not induce stress, anxiety, lead to humiliation or cause harm or any other negative consequences beyond the risks encountered in the participant's day-to-day lives.

Yes

Please include any further relevant information for this section here:

Information gathered will be limited to 'did you enjoy the game so-far?' and 'What do you think could be done to improve it?'

Where appropriate, do you have consent for the publication, reproduction or use of any unpublished material?

Not applicable

Will appropriate measures be put in place for the secure and confidential storage of data?

Yes

Does the research pose more than minimal and predictable risk to the researcher?

No

Will you be travelling, as a foreign national, in to any areas that the UK Foreign and Commonwealth Office advise against travel to?

No

Please include any further relevant information for this section here:**If you are to be working alone with vulnerable people or children, you may need a DBS (CRB) check. Tick to confirm that you will ensure you comply with this requirement should you identify that you require one.**

Yes

Declaration: Please tick to confirm that you have completed this form to the best of your knowledge and that you will inform your department should the proposal significantly change.

Yes

Please include any further relevant information for this section here:

C. Code Samples

1. Player Movement

This function is copied from the Player Movement script in the Unity Survival Shooter tutorial [4]. It takes in a horizontal and vertical value and makes a vector out of them.

```
// void Move(float h, float v) {
    movement.Set (h, 0f, v);
    movement = movement.normalized * currentSpeed * Time.deltaTime;
    playerRigidbody.MovePosition(transform.position + movement);
}
```

2. Player Rotation

This function is copied from the Player Movement script in the Unity Survival Shooter tutorial [4]. It gets where a ray cast from the mouse hits the floor and converts it into a vector that is used to rotate the player to face the mouse.

```
// void Turn() {
    Ray camRay = Camera.main.ScreenPointToRay (Input.mousePosition);
    RaycastHit floorHit;
    if(Physics.Raycast (camRay, out floorHit, cameraRayLength, floorMask))
    {
        Vector3 playerToMouse = floorHit.point - transform.position;
        playerToMouse.y = 0f;
        Quaternion newRotation = Quaternion.LookRotation (playerToMouse);
        playerRigidbody.MoveRotation (newRotation);
    }
}
```

3. Player Health

This function is modified from the Player Health script in the Unity Survival Shooter tutorial [4]. It removes an amount from the players health, then displays this by removing one of the players health units.

```
// public void TakeDamage(int amount) {
    currentHealth -= amount;
    batSwing.ClearBalls();
    if (currentHealth <= 0 && !isDead)
    {
        Death();
    }
    else if (currentHealth <= 1)
    {
        playerHealthUnit3.enabled = false;
    }
    else if (currentHealth <= 2)
    {
        playerHealthUnit2.enabled = false;
    }
    else if (currentHealth <= 3)
    {
        playerHealthUnit1.enabled = false;
    }
}
```

4. Bat Swing

The bat swing gets every ball that is currently in a collider in front of the player and applies a new velocity to each of them. It also sets each balls state so that it will not hurt the player next time they collide.

```
// private void ApplyNewVelocity() {
    if (Input.GetButtonDown("Fire1") && !animator.GetBool("IsGuard"))
    {
        foreach (Collider o in ballColliders)
        {
            Rigidbody ballInstance = o.gameObject.GetComponent<Rigidbody>();
            ballInstance.velocity = newballSpeed * BatArea.forward;
            BallCollisions ballColliderInstance = o.gameObject.GetComponent<Ball-
Collisions>();
            ballColliderInstance.ChangeBallState(1);
        }
    }
}
```

5. Bat Guard

The bat guard switched the guard collider between enabled and disabled dependent on if the current animation is of the player guarding.

```
// void Update() {
    if (animator.GetBool("IsGuard") != guardState)
    {
        guardCollider.enabled = !guardCollider.enabled;
        guardState = animator.GetBool("IsGuard");
    }
}
```

6. Enemy Movement

This function is modified from the Enemy Movement script in the Unity Survival Shooter tutorial [4]. It sets the destination of the nav mesh agent dependent on how close the player is.

```
// void Move() {
    if (playerHealth.currentHealth > 0 && enemyHealth.currentHealth > 0)
    {
        if (isPlayerTooClose == true)
        {
            //move away from the player
            backwards = transform.position - transform.forward * 6;
            nav.SetDestination(backwards);
        }
        else if (isPlayerWithinRange == true)
        {
            bool direction = IsLeftOrRight();
            //move to the side
            if (direction == false)
            {
                //move right
                sideways = gameObject.transform.position + gameObject.trans-
form.right * 6;
            }
            else if (direction == true)
            {

```



```

        //move left
        sideways = gameObject.transform.position - gameObject.trans-
form.right * 6;
    }
    nav.SetDestination(sideways);
}
else
{
    // move towards the player
    nav.SetDestination(player.position);
}
}
else
{
    // Stop enemy from moving
    nav.enabled = false;
}
}
}

```

7. Enemy Health

This function is modified from the Enemy Health script in the Unity Survival Shooter tutorial [4]. It is functionally the same as the Player Health script.

```

// public void TakeDamage(int amount) {
    if (isDead == true)
    {
        return;
    }
    currentHealth -= amount;
    if (currentHealth <= 0 && !isDead)
    {
        Death();
    }
    else if (currentHealth <= 1)
    {
        enemyHealthUnit5.enabled = false;
    }
    else if (currentHealth <= 2)
    {
        enemyHealthUnit4.enabled = false;
    }
    else if (currentHealth <= 3)
    {
        enemyHealthUnit3.enabled = false;
    }
    else if (currentHealth <= 4)
    {
        enemyHealthUnit2.enabled = false;
    }
    else if (currentHealth <= 5)
    {
        enemyHealthUnit1.enabled = false;
    }
}
}

```

8. Enemy Shoot

This function is modified from the Tank Shooting scripts function Fire() in the Unity Tanks tutorial [5]. When the player is in contact with a trigger collider, a new projectile is created in front of the enemy and launched forward. It also sets the balls state so that it will not hurt any other enemies.

```
// void CreateBall(){
    if ((currentNumberOfBalls < maxNumberOfBalls) && (isPlayerWithinRange == true)
    && (playerHealth.currentHealth > 0))
    {
        Rigidbody newBallInstance = ballManager.CreateNewBall(fireTransform.posi-
        tion, fireTransform.rotation);
        newBallInstance.velocity = ballSpeed * fireTransform.forward;
        BallCollisions ballColliderInstance = newBallInstance.gameObject.GetCompo-
        nent<BallCollisions>();
        ballColliderInstance.ChangeBallState(2);
    }
}
```

9. Enemy Spawner

This function asks the enemy manager to spawn an enemy at a given location. The enemy manager is used as a factory so that the amount of enemies in the scene can be easily tracked.

```
// public void CreateEnemy() {
    if (currentNumberOfEnemies < maxNumberOfEnemies)
    {
        Rigidbody newBallInstance = enemyManager.CreateNewEnemyType1(fireTrans-
        form.position, fireTransform.rotation);
    }
}
```

10. Enemy Spawner Manager

This function is modified from the Enemy Manager script in the Unity Survival Shooter tutorial [4]. Instead of directly creating an instance of an enemy at a transform point, it gets code attached to the spawner to do it.

```
// void SpawnEnemy() {
    if (playerHealth.currentHealth <= 0)
    {
        return;
    }
    int spawnPointIndex = Random.Range(0,enemySpawnPoints.Length);
    spawner = enemySpawnPoints[spawnPointIndex].GetComponent<EnemySpawner>();
    spawner.CreateEnemy();
}
```

11. Enemy Manager

This function creates a new instance of a type 1 enemy before adding it to a list and returning the instance to a spawner. The list is used to keep track of how many enemies are currently spawned in the scene.

```
// public Rigidbody CreateNewEnemyType1(Vector3 position, Quaternion rotation) {
    Rigidbody newEnemyType1Instance = Instantiate(enemyType1Object, position, ro-
    tation) as Rigidbody;
    enemyInstances.Add(newEnemyType1Instance);
    return newEnemyType1Instance;
}
```

```
}
```

12. Ball Manager

This function operates the exact same as the Enemy Manager, but instead used to create projectiles. The list is used for the same reason as in the Enemy Manager, to keep track of how many projectiles are currently spawned in the scene.

```
// public Rigidbody CreateNewBall(Vector3 position, Quaternion rotation) {
    Rigidbody newBallInstance = Instantiate(ballObject, position, rotation) as
Rigidbody;
    ballInstances.Add(newBallInstance);
    return newBallInstance;
}
```

13. Main Menu Manager

This function operates almost identically to the Enemy Spawner Manager. Instead of enemies though, this one spawns projectiles. This is used on the main menu to give the users something interesting to look at.

```
// void Update() {
    if (Timer < Time.time) {
        int spawnPointIndex = Random.Range(0, ballSpawnPoints.Length);
        currentBallSpawner = ballSpawnPoints[spawnPointIndex].GetComponent<BallSpawnerMenu>();
        currentBallSpawner.CreateBall();
        SetRandomTimer();
    }
}
```

14. Start Menu Manager

This function is only used to hide the opening message and display the main menu when the user presses any key. The reason for having a start screen is that it is required by console manufacturer as part of the standard Technical Requirements Checklist (TRC) [8].

```
// void Update() {
    if ((Input.anyKey) && (mainMenu.activeSelf == false) && (start.activeSelf ==
true))
    {
        start.SetActive(false);
        mainMenu.SetActive(true);
    }
}
```

15 Pause Menu Manager

This function is copied from the Pause Menu script in the PAUSE MENU in Unity tutorial [7]. This allows the user to pause the game and access the pause menu.

```
// void Update() {
    if (Input.GetButtonDown("Pause"))
    {
        if (isGamePaused == true)
        {
            Resume();
        }
    }
}
```

```

        }
        else
        {
            Pause();
        }
    }
}

public void Pause() {
    pauseMenu.SetActive(true);
    Time.timeScale = 0;
    isGamePaused = true;
}

public void Resume() {
    pauseMenu.SetActive(false);
    Time.timeScale = 1;
    isGamePaused = false;
}

```

16. Game Over Manager

This function is modified from the Game Over Manager script in the Unity Survival Shooter tutorial [4]. When the player runs out of health, the scene is paused and the game over message and menu are displayed. The two addition ifs are to space out the animation for displaying the game over message and the game over menu.

```

// void GameOver() {
    if (playerHealth.currentHealth <= 0)
    {
        animator.SetTrigger("GameOver");
        Time.timeScale = 0f;
        gameOverMenu.SetActive(true);

        if (gameOverSlowDownTimerEnd == gameOverSlowDownRate) {
            gameOverSlowDownTimerEnd += Time.unscaledTime;
        }
        gameOverSlowDownTimer = Time.unscaledTime;
        if (gameOverSlowDownTimerEnd <= gameOverSlowDownTimer)
        {
            gameOverMenuAnimator.SetTrigger("GameOver");
        }
    }
}

```

D. Additional Diagrams

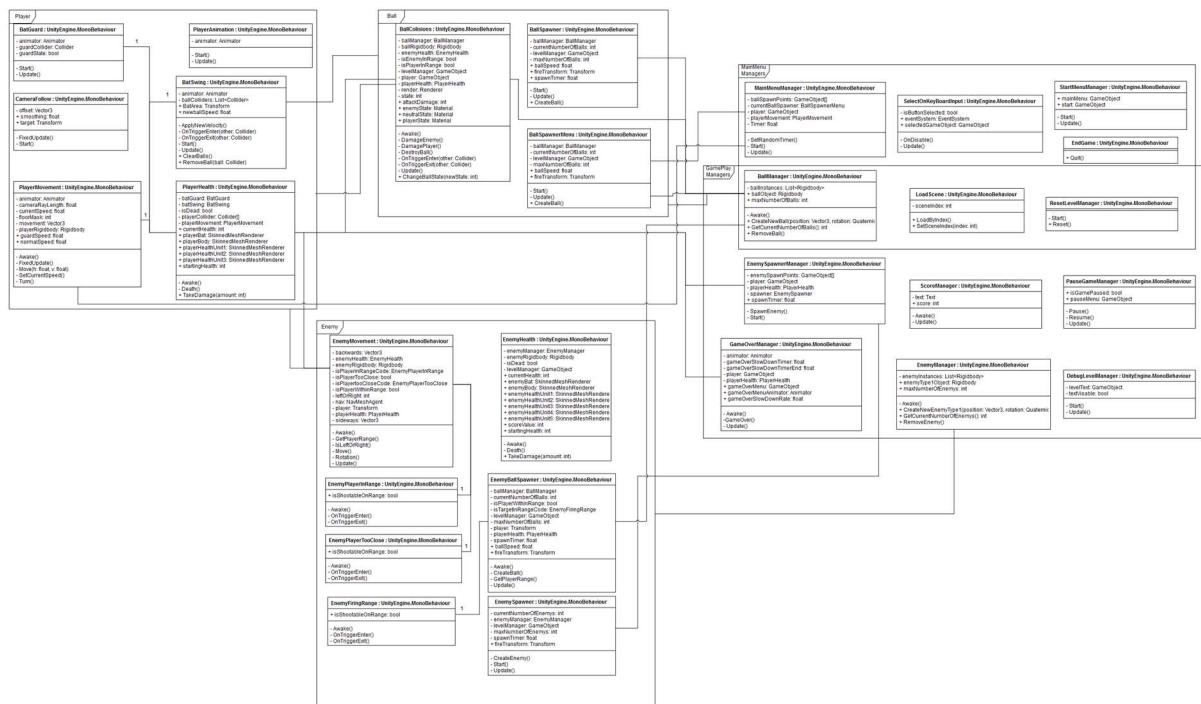


Figure 8: A UML diagram of the program.

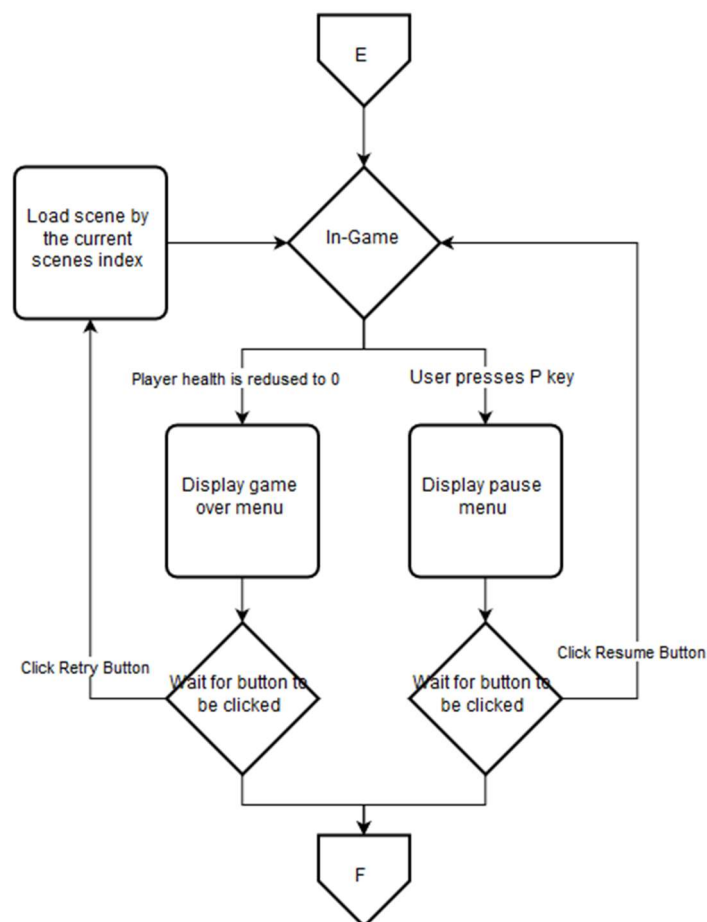


Figure 9: A flow diagram of the pause and game over menus.

E. Test Table

Test Reference	Test Content	Input	Expected Output	Pass Criteria	Pass/Fail
Player – 01	Check that the player can move.	Either the direction keys or WASD.	The player moves in that direction.	The player moves in the expected direction. Result must be repeatable.	Pass
Player – 02	Check that the player can rotate.	Mouse position.	The player rotates to face the mouse pointer.	The player rotates to follow the mouse. Result must be repeatable.	Pass
Player – 03	Check the player can move and rotate simulations.	Mouse position and either the direction keys or WASD.	The player rotates to face the mouse pointer and moves in the given direction.	The player rotates to follow the mouse and moves in the expected direction. Result must be repeatable.	Pass
Player – 04. a	Check the player animates properly.	Left mouse button.	The player swings their bat.	The player animates by swinging the bat. Result must be repeatable.	Pass
Player – 04. b	Check the player animates properly.	Right mouse button.	The player moves the bat in front of them.	The player animates by setting the bat in front of them for a guard. Result must be repeatable.	Pass
Player – 05	Check that player health is working properly.	Ball object collides with the player	The player loses an orange health segment.	The player loses a unit of health. Result must be repeatable.	Pass
Player – 06	Check the bat swing works.	Left mouse button.	Any ball in the bats arc travels in the direction the player is facing.	Multiple balls in the bats arc are given a new velocity. Result must be repeatable.	Pass
Player – 07	Check the guard works.	Right mouse button.	Balls bounces off of the bat. The player takes no damage.	Balls bounces off of the guard collider. Result must be repeatable.	Pass
Player – 08	Check the camera works.	Either the direction keys or WASD.	The camera moves to follow the player to keep them centre.	The camera follows the player. Result must be repeatable.	Pass
Enemy – 01	Check the enemy moves properly.	None.	The enemy moves towards the players location.	The enemy moves towards the players location. Result must be repeatable.	Pass
Enemy – 02	Check the enemy moves properly.	The player is in contact with the	The enemy moves to the side.	The enemy move to either the left or the	Pass?

		outer collider.		right. Result must be repeatable.	
Enemy – 03	Check the enemy moves properly.	The player is in contact with the inner collider.	The enemy moves to the backwards.	The enemy moves to the backwards. Result must be repeatable.	Pass
Enemy – 04	Check the enemy rotates properly.	The players position.	The enemy rotates to face the player.	The enemy rotates to face the player. Result must be repeatable.	Pass
Enemy – 05	Check that enemy health is working properly.	Ball object collides with the enemy	The enemy lose a green health segment.	The enemy loses a unit of health. Result must be repeatable.	Pass
Enemy – 06	Check the enemy shoots properly.	The player is in contact with the mid collider.	The enemy spawns a ball with a velocity.	The enemy spawns a ball with a velocity. Result must be repeatable.	Pass
Enemy – 07	Check enemy deaths work properly.	The enemy has its health reduced to 0.	The enemy is despawned + the score increments by 10.	The enemy is despawned. Result must be repeatable.	Pass
Enemy – 08	Check enemies are spawned properly.	Request from the enemy spawn manager.	A new enemy is spawned in the enemy spawner object.	A new enemy is spawned. Result must be repeatable.	Pass
Enemy – 09	Check enemy spawns are limited.	Request from the enemy spawn manager.	No more than 25 enemies are spawned at once.	No more than the set maximum of enemies are spawned at once.	Pass
Ball – 01	Check ball collides properly.	Collision with player.	Player loses health.	Player loses health. Result must be repeatable.	Pass
Ball – 02	Check ball collides properly.	Collision with player.	The ball is destroyed.	The ball is destroyed. Result must be repeatable.	Pass
Ball – 03	Check ball collides properly.	Collision with enemy.	Enemy loses health.	Enemy loses health. Result must be repeatable.	Pass
Ball – 04	Check ball collides properly.	Collision with enemy.	The ball is destroyed.	The ball is destroyed. Result must be repeatable.	Pass
Ball – 05	Check ball collides properly.	Collision with wall.	The ball bounces off.	The ball bounces off the wall. Result must be repeatable.	Pass

Ball – 06	Check ball collides properly.	Collision with despawner wall.	The ball is destroyed.	The ball is destroyed. Result must be repeatable.	Pass
Ball – 07	Check ball collides properly.	Collision with breakable wall.	The wall is destroyed.	The wall is destroyed. Result must be repeatable.	Pass
Ball – 08	Check ball collides properly.	Collision with guard.	The ball bounces off.	The ball bounces off the bat. Result must be repeatable.	Pass
Ball – 09	Check ball states work.	Request from the enemy ball spawner.	The ball changes to a state where it won't hurt enemies.	The ball bounces off enemies. Result must be repeatable.	Pass
Ball – 10	Check ball states work.	Left mouse button.	The ball changes to a state where it won't hurt the player.	The ball bounces off the player. Result must be repeatable.	Pass
Ball – 11	Check ball states work.	Collision with wall.	The ball changes to a state where it will hurt the player or enemy.	Enemy or player lose health. Result must be repeatable.	Pass
Ball – 12	Check ball spawns are limited.	Request from the ball manager.	No more than 50 balls are spawned at once.	No more than the set maximum of balls are spawned at once.	Pass
GameOver – 01	Check game over works properly.	Players health reaches 0.	The scene pauses.	The scene pauses. Result must be repeatable.	Pass
GameOver – 02	Check game over works properly.	Players health reaches 0.	The game over message is displayed.	The message is displayed. Result must be repeatable.	Pass
GameOver – 03	Check game over works properly.	Players health reaches 0.	The game over menu is displayed.	The menu is displayed. Result must be repeatable.	Pass
GameOver – 04	Check scene switch works properly.	Click Retry button.	The scene is reset.	The scene is reset. Result must be repeatable.	Pass
GameOver – 05	Check scene switch works properly.	Click Exit button.	Change to the main menu scene.	The scene is changed. Result must be repeatable.	Pass
Pause – 01	Check pause works properly.	Press P in-game.	The scene pauses.	The scene pauses. Result must be repeatable.	Pass
Pause – 02	Check pause works properly.	Press P in-game.	The pause menu is displayed.	The menu is displayed. Result must be repeatable.	Pass
Pause – 03	Check pause works properly.	Press P while paused or	The scene resumes.	The scene resumes. Result must be repeatable.	Pass

		click Resume button.			
Pause – 04	Check pause works properly.	Press P while paused or click Resume button.	The pause menu is hidden.	The menu is hidden. Result must be repeatable.	Pass
Pause – 05	Check scene switch works properly.	Click Exit button.	Change to the main menu scene.	The scene is changed. Result must be repeatable.	Pass
Menu - 01	Check start screen works.	Any button.	Hide start message and display main menu.	The menu is displayed. Result must be repeatable.	Pass
Menu - 02	Check menu buttons work.	Menu button.	Hide previous menu, show next menu.	The menus are switched. Result must be repeatable with all menu buttons.	Pass
Menu - 03	Check set scene works	Square arena button	The ready button opens the square arena scene.	The current scene is displayed. Result must be repeatable.	Pass
Menu - 04	Check quit works	Quit button	The application closes.	The application closes. This must not be because it crashes.	Pass