****

**Handheld Games Console**

**Christopher Mahon**

**C12380621**

Bachelor of Science in Computer Science (Infrastructure)

Programme Code: DT211C

**Supervisor: David Carroll**

# Declaration

I hereby certify that the material, which is submitted in this dissertation, is entirely my own work unless otherwise stated.

**Signed: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Acknowledgements

I would like to thank my supervisor, David Carroll, for his dedication and guidance throughout the entirety of this project.

Table of Contents

[Declaration 2](#_Toc447894287)

[Acknowledgements 3](#_Toc447894288)

[Table of Figures 5](#_Toc447894289)

[Chapter 1 – Introduction 6](#_Toc447894290)

[1.1 Project Goal 6](#_Toc447894291)

[1.2 Project Specification 6](#_Toc447894292)

[1.3 Structure of Report 6](#_Toc447894293)

[Chapter 2 – Similar Systems 8](#_Toc447894294)

[2.1 Introduction 8](#_Toc447894295)

[2.2 Overview of Game Genres 8](#_Toc447894296)

[2.2.1 2D and 3D perspective 8](#_Toc447894297)

[2.2.2 Genres 9](#_Toc447894298)

[2.3 Game Concept 9](#_Toc447894299)

[2.4 Similar Systems 10](#_Toc447894300)

[2.4.1 Similar Consoles 10](#_Toc447894301)

[2.4.2 Different Controller Styles 10](#_Toc447894302)

[2.5 Conclusion 11](#_Toc447894303)

[Chapter 3 – Technology Research 12](#_Toc447894304)

[3.1 Introduction 12](#_Toc447894305)

[3.2 Development Methodology 12](#_Toc447894306)

[3.3 Language Research 13](#_Toc447894307)

[3.4 USB Libraries 13](#_Toc447894308)

[3.5 Game Engine Research 14](#_Toc447894309)

[3.6 Game Engine Comparison 14](#_Toc447894310)

[3.7 Conclusion 15](#_Toc447894311)

[Chapter 4 – Design 17](#_Toc447894312)

[4.1 Introduction 17](#_Toc447894313)

[4.2 Game Mechanics 17](#_Toc447894314)

[4.3 Game Rules 18](#_Toc447894315)

[4.4 Use Case 19](#_Toc447894316)

[4.5 Case Design 20](#_Toc447894317)

[4.6 Prototyping 20](#_Toc447894318)

[4.6.1 Software 20](#_Toc447894319)

[4.6.2 Hardware 20](#_Toc447894320)

[4.7 Conclusion 22](#_Toc447894321)

[Chapter 5 – Implementation 23](#_Toc447894322)

[5.1 Introduction 23](#_Toc447894323)

[5.2 Controller Drivers Implementation 23](#_Toc447894324)

[5.3 Game Implementation 25](#_Toc447894325)

[5.4 Conclusion 28](#_Toc447894326)

[Chapter 6 - Testing & Issues 29](#_Toc447894327)

[6.1 Introduction 29](#_Toc447894328)

[6.2 Game Testing 29](#_Toc447894329)

[6.3 Controller Testing 29](#_Toc447894330)

[6.4 Logged Issues 30](#_Toc447894331)

[6.5 Conclusion 30](#_Toc447894332)

[Chapter 7 – Project Evaluation & Conclusion 31](#_Toc447894333)

[7.1 Introduction 31](#_Toc447894334)

[7.2 Project Evaluation 31](#_Toc447894335)

[7.3 Future Work 31](#_Toc447894336)

[7.4 Conclusion 31](#_Toc447894337)

[Bibliography 32](#_Toc447894338)

[Appendix – Technologies Used 33](#_Toc447894339)

# Table of Figures

[Figure 2.2.1 Example of a 2D Perspective [8] 8](file:///C:\Users\Chris\Dropbox\Chris%20Mahon%20=%20Final%20Year%20Project\Documentation\Report\FYPReport.docx#_Toc447894340)

[Figure 2.2.2 Example of a 3D perspective [9] 9](file:///C:\Users\Chris\Dropbox\Chris%20Mahon%20=%20Final%20Year%20Project\Documentation\Report\FYPReport.docx#_Toc447894341)

[Figure 2.4.1: Nintendo Entertainment System Controller Schematic [3] 10](#_Toc447894342)

[Figure 2.4.2: Super Nintendo Entertainment System Controller Schematic [5] 11](#_Toc447894343)

[Figure 2.4.3 Xbox Controller Schematic [6] 11](#_Toc447894344)

[Figure 4.3.1: Health Bar 18](#_Toc447894347)

[Figure 4.4.1: Relational Diagram for the System 19](#_Toc447894348)

[Figure 4.4.3: Use Case 19](#_Toc447894349)

[Figure 4.5.1: Prototype 1 Render 21](#_Toc447894350)

[Figure 4.5.2: Prototype 2 Render 22](#_Toc447894351)

[Figure 5.2.1: ID finder code in Python 23](file:///C:\Users\Chris\Dropbox\Chris%20Mahon%20=%20Final%20Year%20Project\Documentation\Report\FYPReport.docx#_Toc447894352)

[Figure 5.2.2: Reading in the inputs from the controller 24](#_Toc447894353)

[Figure 5.2.3: Changing inputs from the controller when different buttons are pressed 24](#_Toc447894354)

[Figure 5.3.1: Sprite loading code 25](file:///C:\Users\Chris\Dropbox\Chris%20Mahon%20=%20Final%20Year%20Project\Documentation\Report\FYPReport.docx#_Toc447894355)

[Figure 5.3.2: Drawing the sprite onto the screen 26](#_Toc447894356)

[Figure 5.3.3: Gravity function 26](#_Toc447894357)

[Figure 5.3.4: Collision Detection 27](#_Toc447894358)

[Figure 5.3.5: Projectile Sprite Sheet 27](#_Toc447894359)

# Chapter 1 – Introduction

Handheld games consoles have been around since the 1970s and 80s. Throughout the years, these consoles have been extremely popular, however they are often rather expensive and are often very difficult to get the permission and tools for them.

In more recent years, there has been a large movement in the creation of game systems. This visibly seen by the increase in the amount of open source systems such as the Ouya and developer kits for larger systems becoming more readily available.

In this chapter, we shall discuss the goal of the project and clearly write out the specification for this project.

## Project Goal

The intention of this project is to investigate whether it is feasible to create an open source handheld gaming console that is relatively easy to replicate. To achieve this goal, this project will be utilising the Raspberry Pi as the computer for this system. The Raspberry Pi should be usable on battery power.

## Project Specification

The project when completed will consist of multiple components. These is a physical device, drivers that communicate with the controller and the OS, and a game which is used to test the physical hardware.

The physical device is the core of the project, and will be the combination of the Raspberry Pi, the touchscreen, the controller and a power supply all within a case. The Raspberry Pi is the main computer of it, and the user will use the touchscreen to interact with the OS system. The main functions of the controller will be to control a game and other software designed to use it. The console has a fairly simple design, as it consists of a touch screen and a directional pad, four face buttons, 2 shoulder buttons and a start and select button.

The drivers will be how the controller input will be converted into a format that the system and game can understand. It will read in the input according to how the protocol dictates, parse the data it receives and then output it to the OS and any other programs that are listening for it.

The game will be a key component to testing the system to ensure that it functions correctly, so the genre should reflect this.

## Structure of Report

* Chapter 2: Similar Systems

In this chapter, we will be looking at similar systems to the proposed project in order to better understand how this project should work. We will be discussing various different components of the project such as game genres

* Chapter 3: Technology Research

In this chapter, we will be looking at the methodology that will be used in this project, as well as the technologies that will be used for developing the game and the controller.

* Chapter 4: Design

In this chapter, we will be looking at how the software and the casing for this projects were designed and what special considerations needed to be taken when designing them.

* Chapter 5: Implementation

In this chapter, we will be looking at the development process and looking at how each segment of the project was created. We will then be looking at any major obstacles that appeared whilst developing this system.

* Chapter 6: Testing

In this chapter, we will be looking at the testing process of the project. First we will be looking at how the system was tested, and then we will be looking at the results from the tests conducted at each major stage of development.

* Chapter 7: Project Evaluation & Conclusion

In this chapter, we will be looking back at each part of the development and design process and discussing how each component went and reflecting on the project as a whole. We will then be looking at what future work could be done with the project

# Chapter 2 – Similar Systems

## 2.1 Introduction

When attempting to develop any project, it is essential to look at a wide variety of similar systems. In this chapter, we shall examine this project by looking at various different similar systems for the different aspects.

In Section 2.2, we will be looking at the difference between 2D and 3D perspectives in games and the various genres we could develop this game as.

In Section 2.3, we will be looking at different games of the genre we have chosen to figure out roughly what components the game being developed will require.

In Section 2.4, we will be looking at similar systems to the hardware of this project to determine how the project should look and act.

## 2.2 Overview of Game Genres

### 2.2.1 2D and 3D perspective

First we shall be looking at the different perspectives to view the game.

A 2D game is a game that is presented to the user only using 2 dimensions, typically in a top down or side scrolling view. The main advantage of 2D games is that they are rarely performance intensive for the system, however that will depend on the complexity of the game. As well as that, the level of complexity on developing games in 2D is much simpler as it requires less complex maths. The main limitation of 2D games is the movement range, as a player can only ever move in the four cardinal directions: up, down, left and right [7].

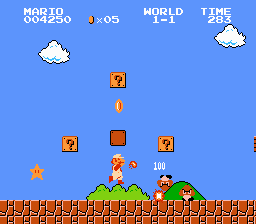


Figure 2.2.1 Example of a 2D Perspective [8]

A 3D game is a game that is presented to the user using the x, y and z axes, typically in a first of third person perspective. The main advantage to using 3D is that you provide the player with a much greater range of movement. The main issue with 3D is that it is much more difficult to work with, as it has a much greater level of complexity then its 2D counterpart [7].



Figure 2.2.2 Example of a 3D perspective [9]

There is much greater support for 2D over 3D on the Raspberry Pi, as it the Raspberry Pi is a relatively weak system and may not be capable of running overly complex 3D graphics without encountering performance issues.

### 2.2.2 Genres

Platformer is a sub-genre of the Action genre. They are most often categorised as the player moving from one location to another through a dangerous environment while trying to complete puzzles or avoiding various obstacles such as holes and enemies [10].   
Although they can exist in both 2D and 3D games, however they are more commonly 2D games. Popular examples of such games are the Mega Man, Super Mario Bros and Sonic the Hedgehog series.

Role-Playing Games (RPGs) is one of the sub-genres of the Adventure genre. This type of game draws inspiration from classical table top games such as Dungeons & Dragons. The typically involve a large quest line that the player is to follow, some form of equipment management system and a levelling system. In more recent years, this genre has evolved to inherently include open world aspects, which allow users to choose how quickly or slowly they progress through the story. Popular 2D games series in this genre include the Final Fantasy and Dragon Quest series [10].

Tactical Role-Playing Games are a hybrid of the Role-Playing Game and the Strategy genres. These games are similar to RPGs in the sense that you control a character or a group of characters, however they incorporate more of a strategic gameplay [10]. The more well-known games of this genre, such as Final Fantasy Tactics and Fire Emblem, use an isometric grid based combat system.

Given that the game will be testing the feel of the system, Platformer would be the obvious choice of genre here considering it is the genre most demanding of responsive controls.

## 2.3 Game Concept

Super Mario Bros is a highly acclaimed game created by Nintendo for the NES. It is a side-scrolling platformer game which consists of the user travelling across a 2D world to defeat the final boss. There is a wide variety of the types of levels that the player must traverse through which include over world, underwater and castle levels. The user must either jump on top or jump over enemies to pass them. While traversing these levels, the player is able to find power ups which help them, like the fire flower which allows the user to shoot a projectile to take out most enemies from a range [8].

Mega Man X is another highly acclaimed game created by Capcom for the SNES. This game is considered a huge success for its well-designed controls and layout. The game is a side-scrolling platformer which consists of 8 highly different levels that can be completed in any order. Once those 8 levels are completed, there is one final level that the user must go through. The combat system is the player shooting projectiles at enemies, similar to the Fire Flower in Super Mario Bros. Each of the bosses of the 8 levels have a weakness to a specific weapon that can be unlocked by killing another one of the bosses, which makes it so that although there is a set path the player is supposed to follow, they are not required to [23].

## 2.4 Similar Systems

### 2.4.1 Similar Consoles

The Game Boy line is a series of handheld consoles created by Nintendo. This line of consoles ran from 1989 to 2005 with the series being discontinued in favour of the Nintendo DS. The latest in this line of consoles was the Game Boy Micro. In terms of the functional hardware, this console was almost identical to its predecessor with the same button layout and a similar under laying hardware. With this system the intention behind it was not necessarily to improve on the performance of the series, but to continue working on improving the comfort of their systems. From a developer stand point, Nintendo consoles are traditionally very difficult to begin developing for as the regulations for getting a developer kit are very strict and very expensive [1].

The Ouya is a now discontinued game console that was released in 2013, that was designed to be an open source platform for developers to make games without needing any specialist equipment or licenses. It also intended to provide free games and software to some extent to all users, either through demos or by supporting in game purchases. [2].

### 2.4.2 Different Controller Styles

The NES used a controller that consisted of 8 buttons. Internally it uses an 8-bit shift register to convert the 8 inputs into the format required to be output into the proprietary connector used by NES controller [3]. See Figure 2.4.1.

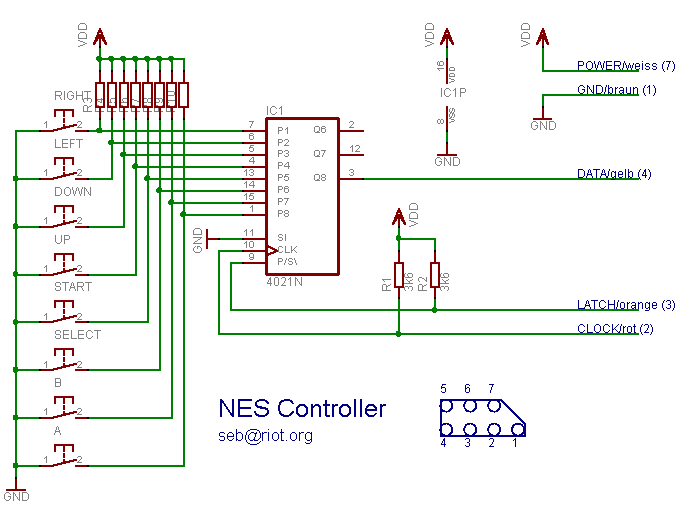
****

Figure 2.4.1: Nintendo Entertainment System Controller Schematic [3]

The SNES controller is an upgraded version of the NES controller. This controller uses 2 shift registers working in tandem to convert the button inputs into the format required for the console to read the input [4] [5]. See Figure 2.4.2.

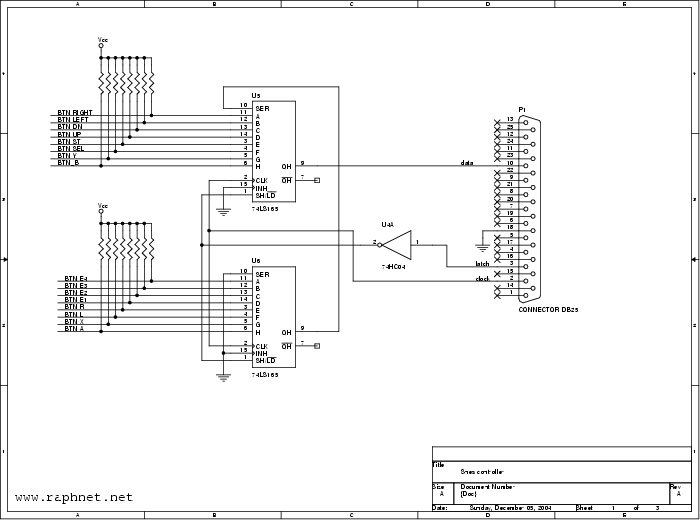


Figure 2.4.2: Super Nintendo Entertainment System Controller Schematic [5]

The Xbox controller uses a relatively similar concept. With the buttons being fed into a device which converts the raw input from the buttons being pressed and outputs data that can be read by the console. In this case because of the greatly increased complexity due to analogue sticks, it’s using a microcontroller rather than shift registers [6]. See Figure 2.4.3.

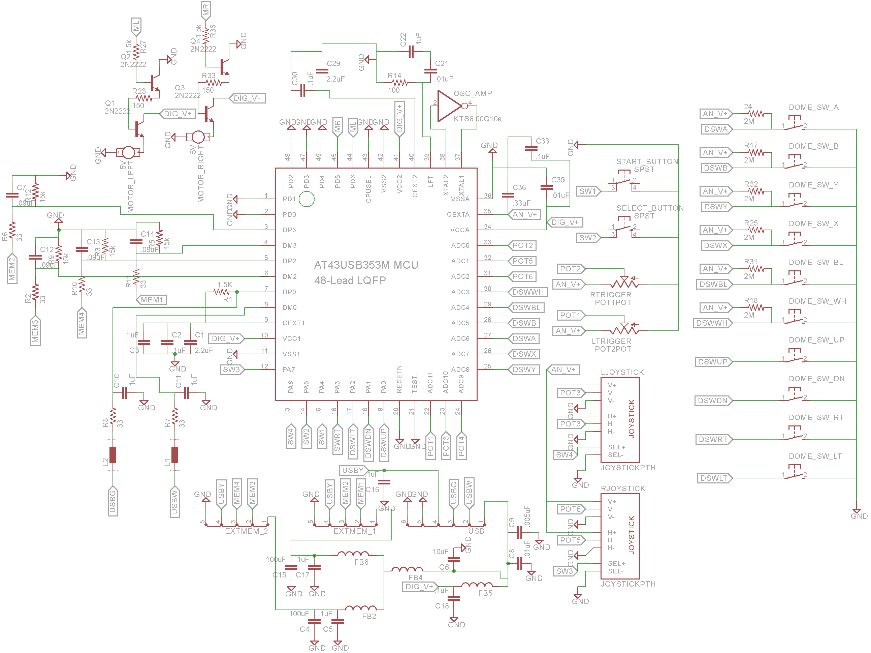


Figure 2.4.3 Xbox Controller Schematic [6]

## 2.5 Conclusion

In this chapter, we looked at various systems that are similar to the system we are developing. In this chapter, we started off looking at different genres that this game could be developed as, as well as looking at the different view perspectives on the game. After having looked at this, we looked at games in the preferred games in an attempt to isolate core components of each genre that this game would need.

In the next chapter, we will be extending our research so far and examine various different technologies that can be used to develop our game.

# Chapter 3 – Technology Research

## 3.1 Introduction

In this chapter, we will be looking at various technologies we can use to develop the controller drivers and the game. Then we will be making a decision on which technologies would best apply to what is being developed.

In Section 3.2, we will look at two different development methodologies and decide on which one would best suit the project.

In Section 3.3, we will be looking into the various languages that are supported by the Raspberry Pi and examine whether they can be used for each deliverable of this project.

In Section 3.4, we will be looking at the available USB libraries that work on the Raspberry Pis for the languages we discuss in Section 3.3.

In Section 3.5, we will be looking at some of the Game Engines available for use on the Raspberry Pi and then we will be comparing them in Section 3.6.

## 3.2 Development Methodology

For this project, two methodologies were researched as potential ways of doing this project. The first of these two is the Rapid Application Development (RAD). The main focus of this methodology is put on development of prototypes rather than focusing on planning. This methodology is emphasises the importance of adjusting requirements as the project grows, based on knowledge gleamed from previous prototypes. The value of this is that the system is forever growing with this type of methodology, and it willing to deter from the initial design if the final result requires it. The main issue with Rapid Application Development is that it requires a huge amount of commitment from the end users as they need to be heavily involved with it throughout the entire life cycle of the project, not just at specific segments. Although not impossible, much larger systems are harder to manage by using Rapid Application Development as there is much less design documentation to follow then other traditional methods.

The other methodology that was strongly considered was the Scrum methodology. Scrum is an agile software development framework, which means that it is a flexible iterative methodology. In Scrum, all work is done in sprints, which are pre-defined time limits. At the beginning of the sprint, the work being done is planned and assigned to various members. Throughout the sprint, the team have regular scrums which are used to detail where members of the sprint are doing well or are falling behind and how any issues can be resolved. At the end of each sprint, there is a review done where the team look back on the past sprint and the work that was and was not completed. Similarly to Rapid Application Development, one of the main shortcomings of Scrum is the requirement of the end user being heavily involved with the system, however they are not quite as involved. Scrum needs a high level of commitment from all team members, or else the project will often not progress as quickly as it needs to be.

Both of these methodologies were considered as they both emphasise flexible requirements and both are inherently capable of dealing with unforeseen issues that appear through development. These components were important as game development is not a rigid process and may require frequent updates of requirements, depending on new information that becomes available. Scrum was chosen as the methodology as it best filled the requirements and limitations of the project and also helped to focus the workflow of the project. Using the sprints in Scrum, it would be possible to deliver small goals regularly, allowing for progress to be gauged easier than normal.

## 3.3 Language Research

Python is an interpreted programming language, which means that programs written in it are able to be run without requiring the user to compile the program into an executable format, although it is still possible to create an executable by using third party software. It is a general use programming languages, which means that it has a large number of practical uses, and has a wide variety of libraries available for it. As Python is an open-source language, it has a large amount of third party libraries available for it.

C+ is a complied programming language which means that when written, the file then needs to be converted to an executable to be able to be run. C++ is a versatile language, and can be used on a variety of platforms such as Windows and Unix-based systems and it has a large amount of support as a result [13].

Java is another compiled programming language that is compatible with the Raspberry Pi, which runs inside of the Java virtual machine which is available for almost every platform. The main reason for this is so that a single compiled file can be deployed on almost any system without needing to be recompiled, as the variations in the systems are handled by the virtual machine itself.

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirements** | **Python** | **C++** | **Java** |
| Graphical libraries available | Allegro, PyGame | Urho3D, Allegro | Lightweight Java Game Library |
| USB libraries available | PyUSB | Uspi | Libusb |
| Relative difficulty (Due to experience etc) | Easy | Medium | Medium |
| Complied language | No | Yes | Yes |

Table 3.3.1: Possible Languages

## 3.4 USB Libraries

For USB programming, Python has a library called PyUSB. This is an open source library written for Python, designed to reduce some of the issues associated with programming USB interfaces [11].

There is a specifically made library called Uspi that is designed to programming USB interfaces in C and C++. This is an open source library that is currently in development which supports a few different interfaces and different devices such as gamepads, keyboards and Ethernet controllers [14].

For USB development, there is a wrapper for Java called Libusb Java. The purpose of this is to allow the use of the Libusb libraries within Java, and is compatible with any devices that Libusb is compatible with [15].

## 3.5 Game Engine Research

In this section, we shall cover a variety of game engines and graphical libraries that could be used to develop the game component of this project.

Unity is a hugely popular game engine that supports both 2D and 3D game development, with separate physics engines dedicated to each one. It is a relatively new emergence into the market and has gained huge popularity with the game development community, resulting in it now being supported by game systems such as the PlayStation 4, Xbox One and Wii U [19].

Urho3D is an open source game engine that can be deployed on the Raspberry Pi. Urho3D can be used for both 2D and 3D game development. Games can be written in with it using C++ and has an integrated 2D physics engine, although it may be required for the user to implement their own improvements on the physics engine depending on their needs. Games created using this engine can be deployed to a multitude of platforms such Windows, Raspberry Pi, Android and iOS with little to no refactoring. Urho3D supports various input devices such as keyboards, joysticks and touch screen input [17].

It would be possible to develop a custom game engine for the game to run on. This would involve mostly creating custom functions for key components such as collision detection and gravity. Although this would be a more time consuming approach and is more susceptible, it is also the most flexible option as each function would be able to work exactly how it needs to. This approach would most likely need the accompaniment of a graphical library.

Allegro is a 2D graphical library that can be deployed on the Raspberry Pi. Allegro is a library that only deals with 2D graphics, however it can be implemented alongside a 3D library such as OpenGL. This library supports various languages such as C++ and Python. As it is a graphical library, it will require a game engine to run alongside it.

Lightweight Java Game Library is an open source Java library that give access to popular APIs such as OpenGL and OpenAL. This game library was used in popular games such as Minecraft by Mojang and is implemented by some game engines [16].

PyGame is a graphical library available for Python. This library is intended as a simple library for novices and experts alike to pick up and implement with ease [12]. As the library is intended for novices, the community is very active and open to helping new developers debug and improve code.

## 3.6 Game Engine Comparison

Now we shall examine the three main engines and compare them to determine which may be the ideal solution for this project. All three engines have built in physics management, meaning that no external packages are needed to deal with gravity and object collision.

Both Unity and Urho3D have a built in editor, meaning that not every single object in the game world needs to be directly programmed to function as desired. Although the Custom Engine will not have a built in editor, since it is being custom designed, it should be fairly easy to implement the engine without requiring an editor.

Since the custom engine is being designed to fit the target platform, it can be programmed in C++, Python or Java depending on which graphical library is being used. Urho3D can be programmed in C++ and Unity uses C# and Javascript.

Unity is the fastest of these engines to implement, as the Unity editor is simple and intuitive to use and has a huge amount of support from the community, it allows for the creation of games to be very quick compared to the other two. Urho3D would be relatively slow to implement a 2D game in as there is very little community support for the engine, and it is not particularly intuitive. The custom engine would be the slowest of these engines to implement as the developer must create and implement the functions of the game rather than relying on pre-built functions.

Compatibility with the Raspberry Pi is one of the more important points to consider when deciding on an engine for this game since it will be deployed solely on the Raspberry Pi. Both Urho3D and the custom engine are compatible with the default OS of the Raspberry Pi. Unity is only capable of running on the Raspberry Pi by using a custom OS.

|  |  |  |  |
| --- | --- | --- | --- |
| **Requirements** | **Unity** | **Urho3D** | **Custom Engine** |
| Open-Source | No | Yes | Yes |
| Supported Languages | C#, Javascript | C++ | C++/Java/Python |
| Physics Management | Yes | Yes | Yes |
| Built-in Editor | Yes | Yes | No |
| Implementation Speed | Fast | Slow | Slowest |
| Compatible with Raspberry Pi | No | Yes | Yes |

Table 3.5.1: Game Engines Comparison

## 3.7 Conclusion

In this chapter we looked at the various technologies that could be used to develop the various aspects of this project.

Initially, the game was going to be created using Urho3D as it supports C++ by default and had support for a variety of different platforms. However after using it for several weeks, it became apparent that it wasn’t what was needed for sufficient progress to be made. The engine was very difficult to set up, requiring the user to download and compile the engine from scratch with very few indicators as to whether it installed correctly or why it doesn’t load. Once loaded, it became apparent that although supported, 2D games were not a main focus of the engine as it was incredibly difficult to create a 2D project and there was very little support for it online. After a few weeks of very little process with Urho, the decision of which engine to use was under consideration again.

After examining the main options again with this new information, the decision was made to go ahead with creating a custom engine. While planning the development of this engine, several of the previous components were re-examined to make sure they were still feasible with this shift in approach. It was at this point that the decision was made to integrate the controller to work internally as a part of the engine rather than run independently of the game. This was done to reduce the overall strain on the OS.

Now that we have an idea on which technologies we will be using, in the next chapter we will begin talking about how to design the hardware and the software.

# Chapter 4 – Design

## 4.1 Introduction

In this chapter we will be discussing how the various components have been designed for this engine.

In Sections 4.2 and 4.3, we will be looking at how the internals of the game will work and how the user will interact with the characters within the game.

In Section 4.4, we will be looking at how each component of the system interacts with the other parts and the possible actions the user will be able to make on the finished system.

In Section 4.5, we will be examining the various requirements that need to be will need to be considered during the design process of the Case.

In Section 4.6, we will be discussing the different prototypes for both the hardware and the software deliverables.

## 4.2 Game Mechanics

The core mechanics that the player will be introduced to immediately while playing will be moving, jumping and shooting. These three actions are the basis for all interaction within the game. From these mechanics, the player has access to a greater range of actions such as damaging enemies, taking damage and collision with other objects. Because all other gameplay components revolve around these three mechanics, it was important to design them properly from the beginning.

In this game, the player is able move around the screen by pressing the Left and Right arrow buttons on the controller. Because it is a core mechanic, it was important to design this feature to be as responsive as possible and to help the player traverse each level smoothly. Because of this, it was important to design the movement functions so the player is able to move at a comfortable speed and to not move to slowly that it becomes arduous to play.

Jumping is another core mechanic of this game that can be triggered by pressing the A button. Its purpose is to give the player a greater range of movement. Similarly to movement, it was important to design this mechanic so that it is pleasant for the player to use. For this mechanic, it was important to recognise any other components and game rules that may interact with it and to account for them. The main component that needed to be considered while designing this mechanic was the impact of gravity on the player character. Because gravity is a force that is constantly being applied to the character, it would important to generate such an upward force that would offset the impact of gravity on the player. As well as that, it was important to restrict the length of time the player would be able to jump for, as otherwise holding the A button would result in the player constantly rising upwards indefinitely. This could be counteracted by creating a timer that begins whenever the player presses the A button, and ends when either the timer runs out or A is released.

Shooting is the main way that the player is able to interact with other characters in the scene. This mechanic is triggered by pressing the B button. When designing this feature, it was important to place restrictions on the characters by making it impossible to shoot a projectile within a certain amount of time since the previous shot. This would be accomplished by creating a timer once a projectile has been shot, and prevent any more projectiles from being created until this timer ends. This would prevent any character to be able to repeatedly shoot projectiles and potentially make the game too difficult or too easy. Another consideration was how properly orient the projectiles once created and how to identify each individual one. This would be accomplished by the character managing its own projectiles, and then passing its location and orientation to the projectile.

## 4.3 Game Rules

Games in general have a lot of underlying rules that may not be obvious to the player, and platformers are no exception.

The main rules of this game are:

* All characters will lose life whenever hit by opponents or a projectile belonging to that opponent
* When the player loses all of their life or drops below a certain height on the screen, the stage is reset.
* When an enemy loses all of their life, they are destroyed.
* All characters will be effected by gravity
* All characters will stop when they collide with most other objects

All characters within the game have a health value. This value may change depending on what event just occurred and whether the character has collided with any objects. For example, the player character will lose one health if hit by an enemy or a projectile belonging to an enemy. The player’s health value is visually represented on screen by the health bar shown in figure 4.3.1. This value is represented by the red bars, up to a maximum of 9. Once the player is on 9 health, they can’t gain more health until they lose some.

Whenever the player gets hit by an enemy or a projectile, they lose some of their health which is visually represented by the health bar shown in figure 4.3.1. When the number of red bars reaches zero, all of the players life has been lost and so the stage resets. Similarly whenever the player drops off the platforms and goes down below the screen, the player will lose all of their life and the game will reset.



Figure 4.3.1: Health Bar

Similar to the player character, all enemy characters have a health value which will decrease whenever they are hit by the player or one of the player’s projectiles. When the health of an enemy reaches zero, that character is destroyed and they will no longer be able to interact in the game or be drawn on the screen.

All characters in the game are subject to gravity, which means that at all times they are moving downwards towards the bottom of the screen. The only exception to this rule is that if they collide with an obstacle while moving, they must stop moving.

The various enemies in the game will require some form of AI to function. Although they will vary depending on each enemy, most of them will share certain key features. For example, all enemies in the game will share some form of pattern in their movements. For example, there will be enemies that shoot during a certain frame of their animation. This is designed to allow the players predict the actions of enemies and remove some unnecessary difficulty from the game.

## 4.4 Use Case

The system consists of multiple components as seen in Figure 4.4.1. The game controller will be connected to the Raspberry Pi through USB. The input of this controller is fed to the system by the controller drives written in Python. The output of the drivers is passed from the system to the game which will be used to control the game which is written in C++ using a custom engine.

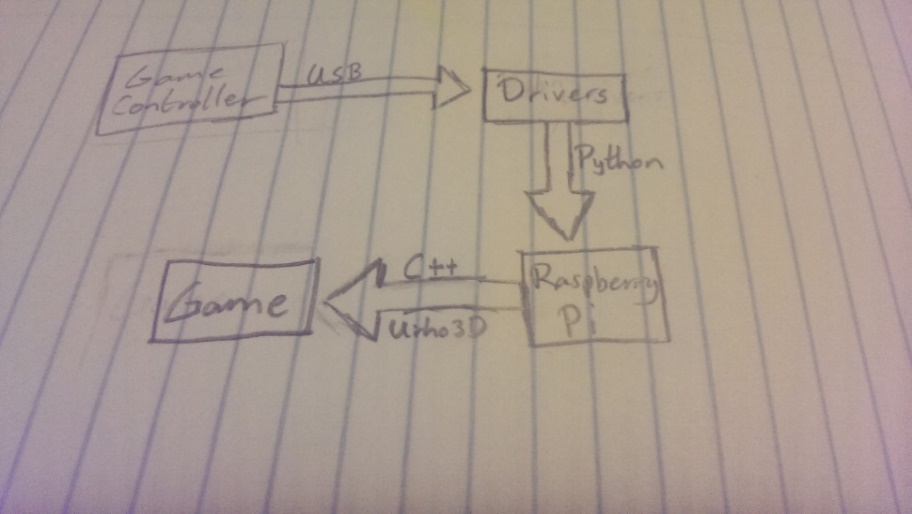


Figure 4.4.1: Relational Diagram for the System

As can be seen from Figure 4.4.3, there is only one type of user that will interact with the system. The player will initially be able to begin the game, configure options and exit the game. After the user begins the game, they will be able to move, jump and shoot their weapon.

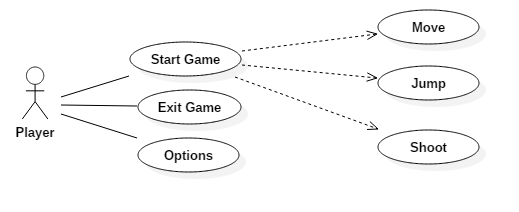


Figure 4.4.3: Use Case

## 4.5 Case Design

There are some usability requirements that needed to be taken into careful consideration when designing the casing of this console. Comfort is a huge concern for this device, as consoles are typically used for extended periods of time. As a result the case would be required to have smooth edges to prevent injury to the user.

Another requirement that is important to take into consideration is that all of the main buttons should be usable from a single holding position. This is to allow users to play games comfortably without having any issues.

The final main requirement for the console is that the weight should be supported evenly enough that the user is able to comfortably able to hold it in one hand. As the console makes use of a touch screen for navigation throughout the system, it is important that the user is able to hold the system in one hand so as to allow the user to navigate without having to place the console on a flat surface.

## 4.6 Prototyping

### 4.6.1 Software

Before developing a game engine, it’s important to know what genre the game will be, and what perspective it will be running this. Once we know this much, it’s important to decide in what core features it will need to function within its genre. The original prototype for this project was done in Unity, to determine the functionality the engine would need. This prototype had basic functionality, in that the player is capable of moving horizontally, jumping and interacting with other objects. Using this prototype, it became apparent that strict collision detection and the constant application of gravity would be very important components to get functional very early in the project.

### 4.6.2 Hardware

The design for the case went through a few different iteration. The first main prototype as seen in Figure 4.5.1 was created originally before any of the measurements for the parts were known, so it was created to give a rough idea of how the system might look and feel. From viewing the case after having been designed, it became very apparent that there were significant flaws with the design. The main among these was that the width of the system was too large, and would just make holding it uncomfortable. Although considerations were made towards making sure the case was rounded so there would be no sharp edges, it did not take into consideration that holding a flat object like this would be particularly uncomfortable.

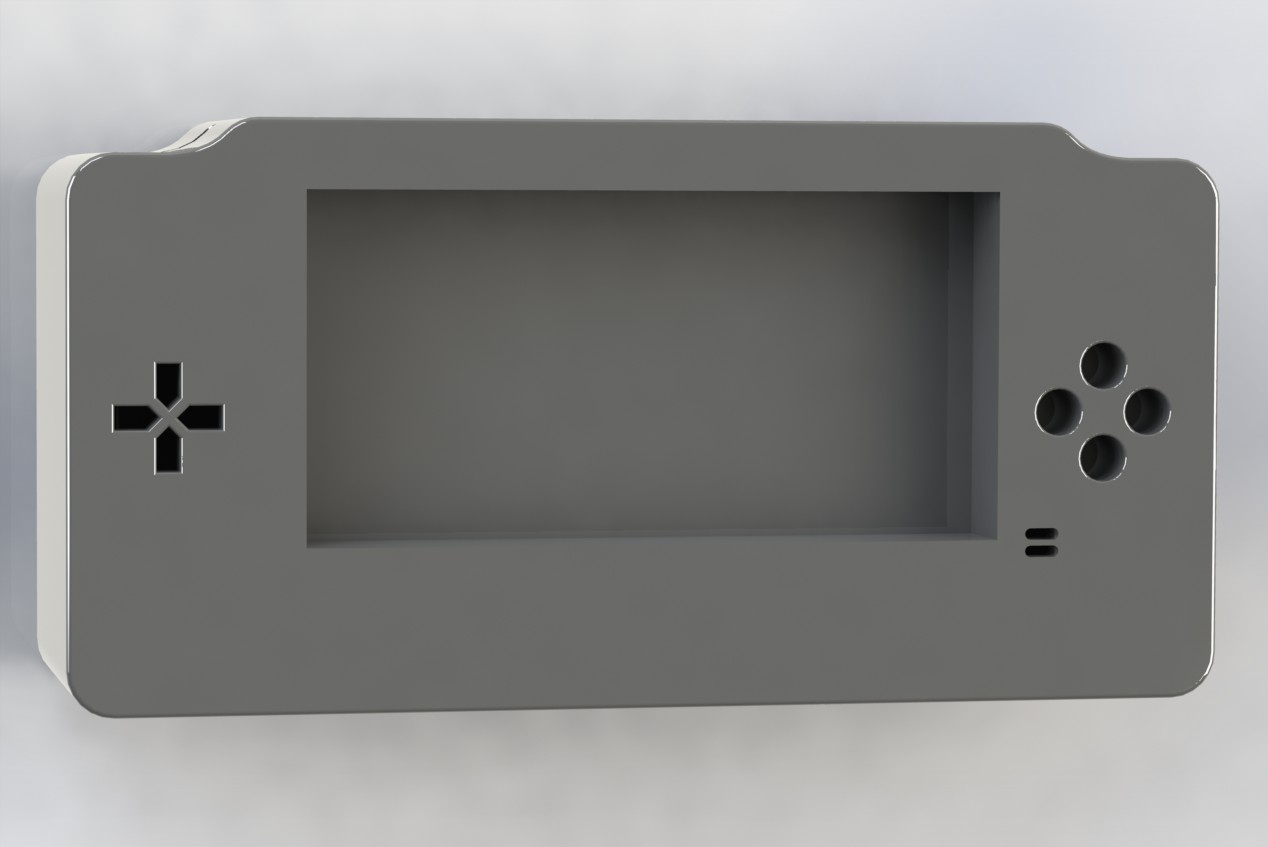


Figure 4.5.1: Prototype 1 Render

Knowing these issues, work began on a second main prototype. This second prototype, as seen in figure 4.5.2 was designed with the dimensions in mind, making it much easier to develop a small device. The main focus on this prototype was to reduce the width so as to remove some discomfort from using it and to adjust the shape so it is not as flat as prototype 1. In prototype 1, the main cause of the width being so large was the touchscreen being in between the buttons on the face of the case. This was solved by shifting the touchscreen upwards and then tilting the screen towards the user. Tilting the screen towards the user allows for the system to be held at a more natural angle without much strain to view the screen properly.

To allow the user to get a better grip on the system and to help position their fingers to the buttons, two protrusions from the bottom of the controller were added on either side. These protrusions are relatively rounded to increase the comfort in holding the system whilst using it. In these protrusions are where the battery boxes are stored. This is to act as a counterbalance to the screen being pushed further up the case and to reduce user fatigue while holding the system.



Figure 4.5.2: Prototype 2 Render

## 4.7 Conclusion

In this chapter, we talked about the various different components that needed to be designed for this project. We looked at design and prototyping in relation to the game and console case and discusses any issues that may have discovered through the prototyping process.

Now that the prototypes have been finished, in the next chapter we shall begin to implement the system.

# Chapter 5 – Implementation

## 5.1 Introduction

In this chapter we shall discuss how the individual components of the project were implemented and integrated together. We shall be going through how each of the components were developed and what complications arose while in developed.

In Section 5.2 we will look at how the Controller Drivers were set up. We will look at the steps taken whilst finding the input of the controller and turning the input from the controller into a format that can be easily read.

In Section 5.3 we shall discuss the initial set up of Urho3D, the switch to a custom engine with Allegro and then the combination of the Controller code and the engine.

## 5.2 Controller Drivers Implementation

To create a program that reads in data from a USB device, finding the right information is key. For the most part, the only way to find this information is trial and error and learning how it reacts to certain events.

When programming the controller, the first main step was finding the identifier for the controller that distinguishes it from all of the other devices. This identifier comes in the form of a Vendor and Product ID. These values are used to ensure that data is intended for the controller makes its way to the correct destination rather than confusing another device. In figure 5.2.1, there is some code written in Python intended on finding the IDs of all devices plugged into the computer.

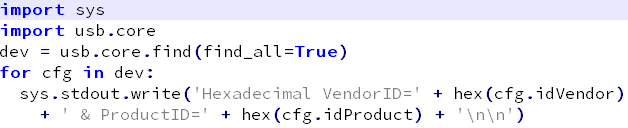


Figure 5.2.1: ID finder code in Python

This code can be used to find the IDs of all the devices connected to a machine. Line 3 calls a function which reads in all the devices connected on the machine, and the following line loops through each device and prints out the Vendor and Product IDs. By running this code twice with the controller plugged in and unplugged, the IDs that belong to the controller would be the one that only appears once. For the controller, the hexadecimal Vendor ID was “0x81f” and the hexadecimal Product ID was “0xe401”.

Once the Vendor and Product IDs are found, the next step was to find the input from the controller. Now that we have the IDs required, we can pass these into the find function used in the previous step to create a connection to the device. Once we have connection, we set the amount of bytes to read in and tell it to read. In figure 5.3.2, we have sample code to set a connection and read the inputs.

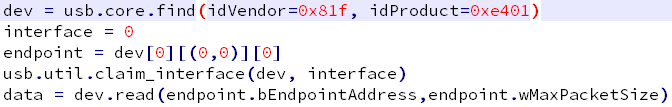


Figure 5.2.2: Reading in the inputs from the controller

In line 1 of figure 5.2.2, we set up a connection to the controller using the Vendor and Product IDs. Then we tell the program how much data we will be reading. On line 4, we are claiming the controller which means that no other device on the machine is capable of taking control of it while we maintain the connection. Line 5 is where we read in the data, we read the data from the controller which we can then either print to the screen or manipulate. Before we close the program, it is important that the device is unclaimed so as to allow other programs to use it.

Once we can see what the controller is sending to the machine, then the next step is to figure out what the inputs mean. This can be done by creating a loop which reads in the data every second and then by observing what numbers change, we can learn which buttons correspond to which value. In figure 5.2.3, we can see the different values changing for the different buttons. For up to 2 buttons that use the same byte, the combination value is mathematically linked, i.e. when A and B are pressed, we get the value 111 on the sixth bit which is the value of A added to the value of B with the default value of 15 taken away. Once the number of buttons pressed exceeds two, the combination values become less predictable. It’s important to ensure that each different combination is dealt with properly, as failing to have all of them would cause in the controller seeming becoming unresponsive during use if too many linked buttons are pushed at the same time. To prevent this becoming an issue, there is a function to take each combination and to set a value in a list to indicate which buttons have been pressed.

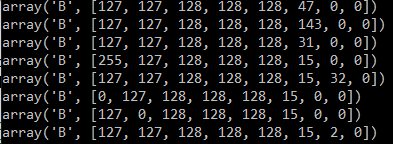


Figure 5.2.3: Changing inputs from the controller when different buttons are pressed

Once the data was in a logical format, the final step was to organising some way of passing these values into the C++ game, so they can be used to steer the player character. After researching the various APIs, the one that was implemented was the C/Python API as it comes pre-packaged with Python, meaning that no additional packages would need to be installed. The important part of using this API is that it should be possible to call upon Python code by use of a function. With the code that was previously written, this was not a possibility as it was all contained within a main function. At this point the code was expanded from a single main function to smaller sub-functions that would all be linked by a central function. This also allowed for easier debugging of individual components as they could be called independently of each other.

## 5.3 Game Implementation

Initially, the game was going to run using the Urho3D engine. After researching the engine, the engine was capable of running on the Raspberry Pi and was capable of making 2D games. This was ideal as they were the two main deciding factors on which engine to use. However during the setup of the engine, there were a few issues that had arisen. The chief among these was that the engine did not install correctly. Whenever the engine was run after setup, it refused to start with the process just ceasing to exist immediately with no warning or error messages. Eventually this issue was solved by downloading a different version of the engine that functioned correctly.

Once the engine was set up and running, another issue was encountered. The UI of the engines editor was unintuitive, meaning that it was incredibly difficult to even attempt to create a basic scene in 2D. After looking online for a solution to the issue, there appeared to be little support for the 2D component of the engine and for the use of the editor in general.

After having spent a couple weeks working on Urho to this point, it became clear that continued use of the engine would not be a valuable use of the limited time allotted to this project. Because of this, the decision was made to swap over to another engine. Given the relative lack of engines available on Raspberry Pi, it was decided to begin work on a custom engine using Allegro as the graphical library.

After having decided to use Allegro for the game component of the project, it was important to set it up on the Raspberry Pi early to avoid any situations like with Urho3D while the project was well under way. Early on in the setup, Allegro appeared to have a few issues running on the touch screen of the Raspberry Pi, as the game window generated by Allegro did not display as intended. After looking for solutions, it became clear that the issue may have been caused by the order that they touch screen drivers and Allegro were installed. At this point, the OS on the Raspberry Pi was reinstalled and then Allegro was installed before installing the touch screen drivers, this order of operations appeared to fix the issue.

Once Allegro was set up, it came time to begin developing the engine. The first step was to create objects and then apply a constant force such as gravity to them. The first component created was the Character object. This would be the core component of the player, enemy and obstacles. In this object, there was a function to load a sprite (See figure 5.3.1) and draw the sprite on screen (see figure 5.3.2).

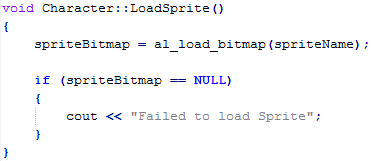


Figure 5.3.1: Sprite loading code

This code when called will load an image into a bitmap from the location supplied to it and then store it within the class to allow for it to be called upon later.



Figure 5.3.2: Drawing the sprite onto the screen

The purpose of this code is to draw a specific part of a bitmap onto the screen in a specific location. The first parameter of this function is the bitmap we loaded earlier, this tells Allegro what it needs to draw. The next two parameters are where in the image the start of the specific part we want is and the two following this are the end of this part. This allows for the game to load a single sprite sheet for an entity and dynamically adjust which portion of it will be drawn. The final three parameters are what tells allegro where to place the sprite and whether to alter the orientation of the sprite. If “ALLEGRO\_FLIP\_HORIZONTAL” was passed into the final parameter it would cause the sprite to flip around the Y axis, allowing for the direction of the player to be visually shown.

Very quickly in the development of the Character class, it became obvious of the need for something to hold the co-ordinates of each object on screen. This lead to the creation of the Vector 2 class. The purpose of the Vector 2 class was to hold the X and Y position of the character and to implement some basic co-ordinate geometry such as a function to find the distance between two points.

Once the Vector 2 class was created, work on getting gravity functioning began. At this point, a gravity function was implemented in the Character class which forced the character to move downwards. Initially this function was designed so as to just modify the position of the character, however this was later modified to allow for easier collision checking (See figure 5.3.3).

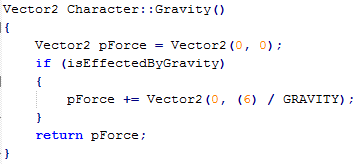


Figure 5.3.3: Gravity function

Now that gravity is implemented, it was time to begin work on detecting collisions between objects. Collision detection can be done in multiple different ways, the method chosen involved drawing an invisible box around the character and the obstacle to check if they overlap. To ensure that it is possible to accurately check that the character will not sink into the object before they move, we project the character forward equal to its force so as to check whether it will move into the obstacle.

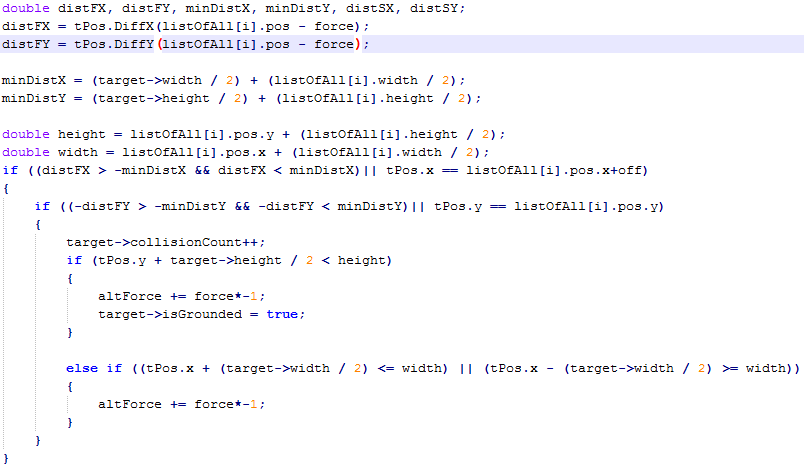


Figure 5.3.4: Collision Detection

The first segment of this code generates the distance between the projected character position and the obstacle on each of axis. It then calculates the minimum distance that should be between the two objects. After it is finished calculating these values, it begins checking whether the distances are outside of the minimum distances between the two objects. If they are overlapping, the force that was projecting the character into the obstacle is negated and returned to be applied to the character. This is called for each obstacle within the level to ensure there is no point of weakness.

Once the Gravity function was working, it was time to implement some form of movement behaviour. This behaviour would be similar to the Gravity function where it would apply some change in the characters location after being checked by the collision function. For testing purposes, walls were placed around the object to ensure that the wall collision checking was functional. At this point, a major error came up during testing. Rather than hitting the wall and stopping as it should, the character kept moving through the wall. It turned out that there was a force being applied to the object that wasn’t being check, so it was able to keep moving forward without being pushed back.

Once the collision detection was functioning as intended, work began on integrating animations. This was done early to avoid any potential errors that may occur during collision detection. In figure 5.3.2, we looked at how the sprites were being drawn on the screen. In figure 5.3.5, we can see an example of a sprite sheet with 4 different frames. In this particular sprite sheet, each frame is an identical size to the others. This is to prevent any issues when attempting to scroll through the sheet.



Figure 5.3.5: Projectile Sprite Sheet

Once the animation infrastructure was in place, it was time to extend the character class into a player and enemy class. For the most part, the player and enemy classes are identical to the character class that they are inheriting from with the main difference being some overloaded functions. This was done intentionally so as to enable the game to process collisions with the player and enemy classes without needing distinct functions for each of them. Once this was implemented, another bug was discovered within the collision detection code. When the function to detect a collision between two characters, anytime a value such as health was being updated, it didn’t actually change. This was caused by the collision detection code receiving a copy of the class rather than a pointer to the object, meaning that any changes were made to the copy and then immediately removed. Also, sometimes when an overloaded function was called from the player class it wouldn’t actually use the overloaded version. This was caused by the player being cast as a Character object rather than a Player object, meaning that it didn’t realise it had an overloaded function. This was fixed by changing a variable that represents a state, and then the function was called whenever it was being treated as a Player object.

Once the objects were acting as they had be prior to the creation, the next component to be added would be the Stage and Level Segment classes. The purpose of the Level Segment is to generate the layout of the stage and then to manage the collision detection with the player and the enemies and obstacles it owns. Each object within the Level Segment stores its location in relation to the Level Segment rather than the world transform. The purpose of the Stage class is to manage the Level Segments on creation, and to manage which segments are loaded at any particular point. This stage of development was primarily making sure the new structure functioned as intended to make it easier in the future to reuse code and to reduce the overall load on the system by making an easy to use structure to restrict the update of irrelevant objects.

The next step taken was to integrate the Python code into C++. This was done by creating a controller class and attaching it to the Player class. The job of the Controller class is to run the Python code using the Python/C API and convert the input the controller outputs into a form that is readable by C++. The player class would then read this data and act accordingly.

## 5.4 Conclusion

In this chapter, we have looked at a step by step run through of the implementation process for both the controller drivers and the game and briefly looked at any issue that may have arisen during this process.

In the next chapter, we will be looking at how the system will be tested. We will then be looking at what bugs appeared during development and a more detailed description of the error and how it was solved.

# Chapter 6 - Testing & Issues

## 6.1 Introduction

For this project, there will be two types of testing conducted: functionality testing and regression testing. Initially, most versions of the game will be tested internally initially, to attempt to find and remove as many bugs as possible. Once the build is satisfactory, it will be handed to external users to be bug tested. This feedback will be collected and used in the creation of the next minor build.

Functionality testing consists of using the piece of software in an attempt to discover issues. Regression testing consists of re-testing a system after previous bugs have been fixed.

In Section 6.2, we will be talking about how the game components will be tested, and the importance of getting external users to test a game with little information about it beforehand.

In Section 6.3, we will be looking at how the controller will be tested.

In Section 6.4, we will be looking at any issues that were discovered from the testing completed at the end of each stage of development.

## 6.2 Game Testing

For testing the game, the external users that will be testing the system are gamers as they would best know how the game should feel and act. Although it would be ideal to get a large group of varied testers to use the system, it may not be possible in the early stages as it is limited to very specific hardware.

It’s important to get large amounts of people testing the system from different viewpoints, particularly in games testing, as there are a huge amount of different combination of things that can create bugs and glitches to happen. For example, there was a bug that was discovered in Super Mario World at the beginning in 2015 that allows players to finish the game in under 3 minutes time [24].

It is important for the testers to ideally as little knowledge of the test cases themselves as that knowledge may influence their thoughts and actions and reduce the potential of getting the system tested from their own unique viewpoint. Because of this, it will be important for the test cases to be filled out personally while observing the tester using the system.

## 6.3 Controller Testing

The controller will be tested in two separate stages. The first stage will consist of directly testing the Python code that reads in the data from the controller. In this stage, the tests being conducted are mostly just to discover whether each input is being read accurately and consistently. These tests will consist solely of holding buttons and observing the changes that occur to the output.

The second stage will consist of testing the controller using the game. What this entails is testing to make sure the controller has no latency issues whilst having the game running. These tests will consist of testing the game using the controller input.

## 6.4 Logged Issues

In this section we will cover discuss the various stages of bug testing where bugs and other issues appeared.

Stage 1 was the testing of the functionality of the controller. The only issue that occurred during this stage of testing was that while running the controller code on a UNIX system such as Raspbian, there was an error preventing the program from taking control of the controller unless it was ran with Root privileges. This was an issue caused by the program attempting to detach the controller from the OS kernel and claiming it for itself. This was an error that was fixed by claiming the device as the user who would be running the program.

Stage 4 was the testing of the movement of characters in the game. The main issue that was encountered during this stage of development was that the characters were moving through the walls when the collision detection should have been preventing this from happening. This issue was caused by the character being moved by a force without it being checked by the collision detection. This bug was fixed by checking this force with collision detection code.

Stage 6 was the testing of the implementation of the Player and Enemy classes. At this stage, we ran into the bug where values, such as health, weren’t updated when they should have been. This was caused by passing a copy of an object into a function rather than passing the pointer to the original.

Stage 8 was the testing of the Controller class. Stage 8 was one of the more difficult stages to debug, as there were a lot of components that seemed to not work. The most obvious of the bugs found in this stage was the game ran much slower then it previously had. This was obviously an issue as it prevented the game from being playable. This was caused by the game creating an instance of the Python interpreter, running the code to read from the controller and then close the interpreter each time the update method happened.

Another major bug that was a little bit more difficult to locate was related to reading the return value from Python. The value that is being returned from Python each time the code is called isn’t being recognised by C++ as being in the correct format. This diagnosis is further supported by the same code functioning correctly when supplied directly with a properly formed string in the expected format.

## 6.5 Conclusion

Testing is an important part of any development process, as is shown by Section 6.4. In this section, it became very clear how important processes such as regression testing is, as two components that could appear to be completely separate, could actually be interfering with each other without noticing. Given the amount of issues that were discovered as a result of the user-based testing, this was made very clear that it was an almost essential addition to the bug testing process.

# Chapter 7 – Project Evaluation & Conclusion

## 7.1 Introduction

In this chapter, we will be evaluating the project and discussing some future work that could be done to improve this project.

In Section 7.2, we will be evaluating the project as a whole.

In Section 7.3, we will be discussing any future work and plans for the system and engine.

## 7.2 Project Evaluation

Although incomplete, the project feels as though it was successful to an extent. This project was an invaluable lesson in how to develop and execute a project plan in the best way possible. The biggest downfall of this project was the time spent using an engine that would become irrelevant at the end. Because there was issues getting the hardware required on time, the prototype was developed using Unity3D, meaning that a significant portion of the time spent on the project was spent learning and using an environment that was known would eventually become irrelevant.

## 7.3 Future Work

In this section we shall look at work that could be done with this project in the future to improve it.

Improvements in the performance between the controller and game would be one of the more important bits of work to do in the future. As it stands, this is a very serious flaw within the system as it currently is.

The current variety of enemies is lacking in the current version and should definitely expanded in the future. As it stands, the player has one enemy that appears in the game and it is just a basic behaviour that shoots a projectile after a certain length of time has passed.

As not every character is animated, the graphics of the game are definitely something that could be worked on in the future.

## 7.4 Conclusion

The main goal of this project was to create a handheld gaming system using the Raspberry Pi. As the project concluded, this goal has been attained. However there are goals that were not met. Another main goal of this project was to develop a game for the system as well, although this goal was not met, most of the ground work to attain this has been accomplished. Once completed, the game engine created in this project could potentially be used to develop any number of 2D games.

Through the development of this project, it became very obvious that the planning and testing stages of game development are some of the more important systems. If a timeline isn’t laid out and adhered too as closely as possible, it can cause some serious issues to the health of the project. Similarly with testing, if rigorous testing isn’t conducted at each major build of the game, it is very likely that bugs will slip past and cause large issues later on.

# Bibliography

1. Sarrel, M, “Game Boy Micro”, <http://uk.pcmag.com/game-boy-micro/26373/review/game-boy-micro>, 2005, ( last accessed 7December 2015)
2. Ouya Inc., “About Ouya”, <https://www.ouya.tv/about/> , 2015, ( last accessed 7 December 2015)
3. Gordon, “NES Controller on the Raspberry Pi”, <https://projects.drogon.net/nes-controller-on-the-raspberry-pi/>, 2012, ( last accessed 7 December 2015)
4. Christy, J. and Game Station X, “Super NES Controller Data”, <http://www.gamesx.com/controldata/snesdat.htm>, 1999, ( last accessed 7 December 2015)
5. Assénat, R., “Arcade style controller for Snes, NES, and PC”, <http://www.raphnet.net/electronique/arcade_control/arcade_control_en.php> , 2014, ( last accessed 7 December 2015)
6. Gilmers, J., Xbox controller schematic [online image], <http://courses.ece.msstate.edu/ece4723/dissect/805529378889/images/xboxControllerSchematic.png>, 2013. ( last accessed 7 December 2015)
7. Stack Exchange Inc, <http://gamedev.stackexchange.com/questions/631/what-to-consider-when-deciding-on-2d-vs-3d-for-a-game>, 2010, ( last accessed 7 December 2015)
8. “Super Mario Bros”, Nintendo Entertainment System, 1985, 13th of September, Nintendo, Nintendo R&D4
9. “Fallout 4”, Microsoft Windows, Playstation 4, Xbox One, 2015, 10th of November, Bethesda Softworks, Bethesda Game Studios
10. Stahl, T., “Video Game Genres”, <http://www.thocp.net/software/games/reference/genres.htm> , 2005, ( last accessed 7 December 2015)
11. Walac, “PyUSB”, <https://walac.github.io/pyusb/> , last updated 2015, ( last accessed 7 December 2015)
12. Pygame Community, PyGame, <https://pygame.org> , 2015, ( last accessed 7 December 2015)
13. Stroustrup, B., *The C++ Programming Language Fourth Edition*, Michigan, Addison-Wesley, 2013
14. Rst, “Introducing USPi – A bare metal USB driver written in C“, <https://www.raspberrypi.org/forums/viewtopic.php?f=72&t=92579> , 2014, ( last accessed 7 December 2015)
15. Ursgraf, “LibUSBJava”, [http://libusb-java.ch/](http://libusb-java.ch/start) , 2015, ( last accessed 7 December 2015)
16. Lightweight Java Game Library Project, “LWJGL – Lightweight Java Game Library”, <https://www.lwjgl.org/> , 2014, ( last accessed 7 December 2015)
17. Urho3D, “Urho3D”, <http://urho3d.github.io/>, 2015, ( last accessed 7 December 2015)
18. Allegro, “Allegro”, <http://liballeg.org/>, 2015, ( last accessed 7 December 2015)
19. Unity Technologies, “Unity – Game Engine, tools and Multiplatform”, <https://unity3d.com/unity>/ , 2015, ( last accessed 7 December 2015)
20. Engadget, <http://www.engadget.com/products/ouya/controller/>, 2013, ( last accessed 7 December 2015)
21. Nzen Mods, “Nzen Mods | eNcade”, <http://nzenmods.com/>, 2015, ( last accessed 7 December 2015)
22. RobotLovesKitty, “Gamekid”, <http://robotloveskitty.com/gamekid/>, 2015, ( last accessed 7 December 2015)
23. “Mega Man X”, Super Nintendo Entertainment System, 1993, 17th of December, Capcom, Capcom
24. “Super Mario World -- Credits Warp in 3:07.2 (Former World Record)”, [online video], 2015, <https://www.youtube.com/watch?v=HxFh1CJOrTU>, (last accessed 8December 2015)

# Appendix – Technologies Used

Languages Used: C++ for the game, Python 3.4 for the controller

External libraries required for C++: Allegro 5.2.0

External Libraries required for Python: PyUSB 1.0

Programs used for Art Assets: Aseprite 0.9.5