

Metal Metal Land

Final Year Project Report

DT228

BSc in Computer Science

**Student: Paul Redmond**

**Supervisor: Mark Foley**

School of Computing

Dublin Institute of Technology

**02/05/2018**



Abstract

“Metal Metal Land” is a procedurally generated, multiplayer action game implemented using the Unity (C#) game engine with the goal of it being as fast paced as possible with as minimal load times as possible[?]. It involves two players selecting characters, each representing different sub-genres of metal (such as Old-School British Metal, Pirate Metal, Thrash Metal etc.) and fighting to the death in a procedurally generated, destructible arena using a variety of weapons, power-ups and environmental objects to do so.

The key goal of this project is to produce a fast paced local multiplayer action game. As technology has improved, the physical interaction between players as they play games with one another has faded away more and more, being replaced with online multiplayer almost entirely. The absence of traditional “couch-based” local multiplayer in modern games has not gone unnoticed and there has been a cry for a return to the local multiplayer games of old from a large section of the gaming community. I am in agreement with this view and while I appreciate the convenience that the modern online-based multiplayer provides, it does leave me wanting for games similar to the ones I grew up with.

When the game boots up, players are presented with the main menu, containing links about how to play the game, game options and the actual play-game button. If players are already familiar with the game and its rules, they should be able to begin playing the game within 10 seconds of arriving on the main menu. Once they begin playing, they will be presented with minimal interruptions so as to not negatively impact the flow of the game. The feedback from initially testing this game showed me that people were very open to, and fans of, the minimal load times and rapid pace of the game.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<Student Name>

<Date>

Acknowledgements

Firstly, I would like to thank my parents who have supported me physically and emotionally over the past few gruelling months. I’d also like to thank the lecturers from DIT who helped with this project, Dr. Susan McKeever, Dr. Bryan Duggan and in particular my own mentor Mark Foley, who helped and supported me to the absolute best of his ability throughout the entirety of the this project.

Additionally, I’d like to thank my classmates; Aaron Renaghan, Eamon Tang, Jay Hynes, Alannah Mullins and Oleg Petcov for all the great times we’ve had over the last 4 years. I’d also like to thank Ally Daly for keeping me plied with burritos, pints and the most adorable dog pictures over the last year and beyond to help keep me motivated.

I’d also like to thank the people who helped me test my project and gave me feedback throughout; Jonathan Bradley, Emily Kremers, Ally Daly, Michael Clonan, Aaron Jennings, Gary Raymond, Liam Phelan, James Hassett, Nicola O’Brien, Max Curtis, Haashim Bari, Ross Murphy, Eamon Tang.

Finally, I’d like to thank all the Metal bands, without whom this wouldn’t be possible, who inspired me to make this game such as “Metallica”, “Alestorm”, “Gloryhammer”, “Black Sabbath”, “GWAR”, “Judas Priest” and the countless other bands that have fuelled my passion for this genre. In particular I’d like to thank “Lagerstein”, who generously allowed me to use their music in the game and most importantly of all, I’d like to thank “Iron Maiden” for introducing me to the genre and for remaining my steadfast favourites over the last 14 years.

# 1: Introduction

Local multiplayer games were originally a mainstay in the gaming industry. Multiple games popularity were defined by the depth and fun brought by their multiplayer experiences, this is despite the fact that some of these games were originally designed with single-player being at the forefront of the project. However, as video game consoles, PC’s and the technology that powers them have advanced, local multiplayer began to fade away in favour of entirely single player experiences or in favour of cloud-based online multiplayer games. This however has not been a transition without problems. While cloud-based online multiplayer games introduce an entirely new level of convenience, allowing players to play games with one another regardless of geographical position, it has brought with it a level of disconnect between the players. Some video game developers, independent developers in particular, have begun to notice the lack of traditional local “couch-based” multiplayer games and the nostalgic cry from the gaming community for a return to these simpler games.

A new technology that can be found in modern games to varying degrees of prevalence is Procedural Generation. Procedural Generation generally refers to the process of generating game content automatically, using a variety of different algorithms to do so. In the case of video games, this usually involves dynamic level generation and difficulty. The benefits of introducing procedural generation into a game is that it generally allows for over-all smaller file sizes while dramatically increasing the content available in the game. This allows for a potentially varied experience with each play-through of a game. Procedural generation in games in the last few years has been used to create objects of almost any form. This includes music, art, world design, enemy behaviour and more.

As previously mentioned with regards to local multiplayer and online multiplayer, technology is forever changing and improving. While this does allow for projects of grander scope and scale to be achieved, a consistent problem with some of these games is the introduction of multiple loading screens. These screens generally break the player’s level of immersion in the game and can take them out of the game.

In the following report, I will discuss my research into each of these 3 aspects that I seek to overcome with my project. Additionally, I will also include my own process into designing and developing the game as a whole and what potential future this project could have.

### 1.1 Project Overview

When the players wish to start the game, they will be greeted with a character selection screen. Each of these characters, while uniquely designed, all share the exact same functionality. The designs of the characters come from different sub genres of metal music including; Power Metal, Thrash Metal, Alt Metal, Pirate Metal, Old-School British Metal, Hair Metal, Death Metal and Metal Mascots. When each player has selected their character they will be brought to an options screen allowing them to have a certain degree of control over the levels and rounds they will play in. These options range from terrain type, which will control the general shape the terrain should adhere to (If it should adhere to any shape in particular at all), sudden death mode, round count and more. When these options are set, the players will spawn in a procedurally generated arena which will include everything they need to defeat their opponents. Depending on the player’s option choices, stage hazards such as explosive barrels or spike pits may appear. A number of weapon spawning altars will spawn in with the players, the number of which is determined by the overall amount of land squares that make up the arena. The players will use these weapons in order to try and defeat the other player with each shot generally equalling a kill if hit. When a player dies, a 1.5 second timer is activated. If the surviving player dies in this time, no point is awarded. However, if that player survives, there are awarded a point. The next level at this point is brought in, ideally with no wait time or loading screen as it should have finished generating while the previous round is in progress.

### 1.2 Project Objectives

The main goal of this project is to deliver a high quality, fast paced, local multiplayer action game. Additionally this game should have the ability to create destructible levels that are dynamically generated based on a very abstract player input (E.g. “Standard”, “Valley”. “Random” and so on) with each level being at a reasonable of playability (for example, a player should not spawn inside a set of blocks or on top of a spike pit). All of this should be accomplished at a rapid pace with minimum load times or intrusions which could potentially impact player immersion. In order to accomplish this, an efficient procedural generation algorithm shall be used alongside a-synchronous level generation (wherein the levels are generated in the background, masking potential load times). The game itself will be developed on Windows using the Unity Game Engine, specifically using C# as its core language.

### 1.3 Project Challenges

One of the challenges that I ran into was trying to find an algorithm which can procedurally generate levels of a reasonable level of quality efficiently. As previously mentioned, it is one of the key objectives of this project to generate these levels with no loading times apparent to the user, this includes the game stuttering, lagging or being adversely affected in some other form. While certain procedural generation algorithms are less complex than other ones, they may not offer the level of quality that I would be looking for in terms of level generation. As such, it is important to find the right balance between level generation efficiency and level generation quality.

Another challenge that I ran into while completing this project was animation. While I have in the past completed projects in Unity, I never under took a project of this scope and never designed my own art or animations for a game before. While initially designing the character and environment sprites, I felt that I was going into more detail than was originally needed and that creating roughly 15 sprites for each character would have wasted far more time than was really warranted. I therefore decided to settle on a more minimalist approach with regards to character and environment art which I feel not only looked a lot better, but saved a lot of time in addition to being a lot easier to work with animation wise.

Another challenge, and arguably the hardest to overcome when building a player versus player game like this, that I encountered was balancing the game to ensure that all weapons and power-ups were equally powerful in an overall sense. For example, the rocket launcher weapon should obviously do more damage than a handgun, however it should also have a directly proportional amount of drawbacks to firing it. For example, in the case of the Rocket Launcher, I set it so that it can only fire at a dramatically slower rate compared to the handgun. The reason why this particular challenge was so difficult was that it was pretty much impossible to test on my own as I would need to be competitively playing against myself using two controllers at the same time, which really isn’t feasible. This required me to test it with friends and class mates quite often, each time making iterative changes based off of both their feedback and things that I noticed myself.

### 1.4 Structure of Document

# 2. Research

In this section of the report, I will explain the research that I undertook prior to beginning development on this game and what other research impacted the game as I was developing it. This includes what other projects I looked into and what impact I feel they had on the design of my own project. In addition to this I will also discuss what technologies were examined and used as part of this project. This includes game engines and procedural generation algorithms.

### 2.1 Project Background Research

When I was originally trying to come up with the basic design of this project, its theme and general gameplay were quite different from what they ended up being. Originally, the game was going to be called “Beat Rush” and would have 1 character from popular genres such as Blues, Pop and Metal. The characters originally were never meant to kill each other particularly quickly also, instead, they would have to damage each other over time using different weapons and moves before trying to finish their opponent off for the kill. The success rate of finishing off another player was based around the amount of damage that player had endured.

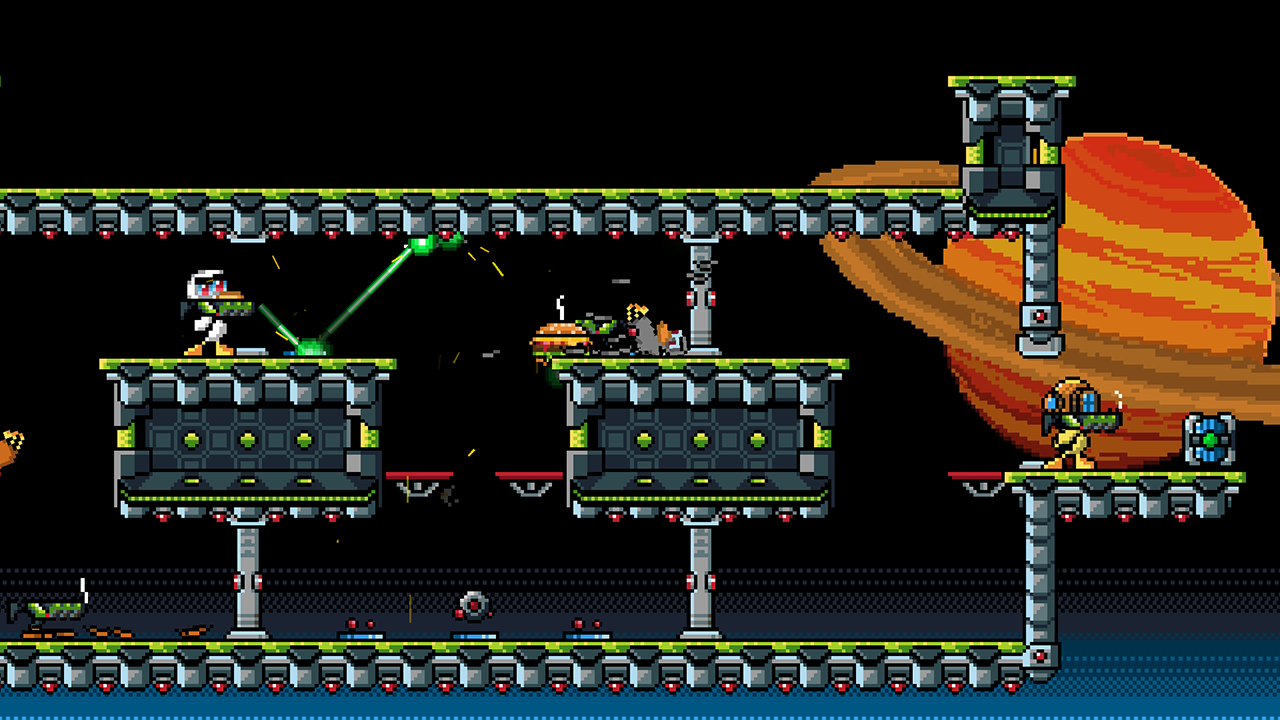
These ideas however were quickly changed when I started to examine other games I liked that would fit into this genre. Rather than rounds or player encounters taking a long time, like in the original version of “Beat Rush”, they would usually have a sort of “Glass Cannon” approach to player versus player reactions, meaning that while players can do a lot of damage to one another, they were also equally susceptible to damage. This glass cannon approach allows for rapid and quick encounters where players are generally able to kill their opponents within a very short space of time. The glass cannon approach allowed for a game to be not only fast paced but more simplistic as well which would allow for people who aren’t usually familiar with the genre to be able to understand and to get used to the general gameplay mechanics quickly.

Additionally, while trying to design characters for “Beat Rush”, I found that I had a lot more fun and a lot more ideas for metal and rock themed characters. I also noticed that when I looked at other games that I enjoyed, the characters generally revolved around one singular narrow theme. For example, in the game “Broforce”, which will be discussed in greater detail later in the report, player characters are based around parodied versions of famous action movie characters such as “John Rambo” and “The Terminator”. Another example of game using a narrow theme for its characters is “Duck Game”, wherein players just play as different coloured ducks.

Before beginning this project, I had only a small bit of previous experience with regards to game development. As part of my Object Oriented Programming module in the 2nd year of my course, I grouped up with two other friends to create a C# based game in the Unity game engine called “Pirates in a Barrel” [1]. Pirates in a Barrel was a tower defence game where players would have to defend a castle from a series of different pirates in different tiers of boats by building towers along the edge of a river. Originally, I struggled with getting certain mechanics to work as it had never something that I had actively attempted to do before. However, as the project continued on, everything began to fall into place and since completing that project, I have tried to familiarise myself further, focusing on the unity game engine in particular.

### 2.2 Alternative Solutions to the Project

One of the pre-existing games that I looked at when researching this project was “Duck Game”. Duck Game is a 2D action game where each player plays as a different coloured, hat-wearing duck. The purpose of this game is to kill other opposing ducks using different weapons and abilities in a serious of previously generated levels. This process repeats until it reaches a pre-determined round count, usually 10. This game was pretty vital for determining some of the base mechanics I settled on in my game.



*Duck Game*

This was one of the first modern indie games that really started to revive my love for local multiplayer action games. The control scheme in Duck Game is exceedingly simple, ensuring that people of all skill levels would be able to play this game at a reasonable level. There are only 5 key buttons required for Duck Game; Jump, Pickup/Drop Weapons, Fire Weapon, Move and Quack. Aside from understanding what weapons were more effective than others and those 5 controls, there wasn’t much else that a user would need to know in order to play this game. This is one feature that I really wanted to try and include in my own game in order to ensure that other players, who may not be used to playing video games like these, would be able to play. One additional aspect from Duck Game that I wanted to try and implement into my own project that was how fast paced it was. Outside of the initial loading screens to get into the actual game, the only other loading screens the players encountered were short (usually 0.5 seconds) loading screens in between each round. While this length is a pretty insignificant amount, as part of this project I wanted to try and remove these loading screens entirely.

One other pre-existing game which is similar to this project is “Bro-Force”, which is developed by “Free Lives Studios” [brforce1]. BroForce is a side-scrolling 2D action game where player’s play, as previously mentioned, as parodies of famous action movie characters. Each of these characters retain the appearance and over-all abilities of that character. Their names however are slightly modified to include the word “Bro” in some form such as “John Rambro” instead of “John Rambo” or “The Brominator” instead of “The Terminator”. The game features both local multiplayer and online multiplayer as well as single player. The players then play through a pre-generated campaign as these characters. One of the key features of this game is that it sports destructible terrain. Depending on a particular characters weapon or abilities, they can destroy the terrain at different rates. For example, the character John Rambro uses a basic machine gun style attack which can slowly destroy terrain in a straight horizontal line from the player. The Brominator on the other hand uses a mini-gun which has the same area of attack but can destroy terrain at a much higher rate when compared to John Rambro. In the interest of fairness and balance however, using the mini-gun causes the player to be pushed backwards at an increasingly higher speed, depending on how long they fire the gun for. With regards to what I chose to adopt from this project, I felt that including a terrain destruction feature would strongly benefit the competitive nature of my game as this could allow players to attempt to whittle away a platform an enemy player is standing on in order to increase their chances of defeating them either by using their own weapons or by removing all available land entirely. Additionally, another aspect I decided to adopt into this game is the multiple different weapon behaviours such as the previously mentioned mini-gun behaviour. The reason for this decision is that I feel it is only logical that a character should react differently to firing a handgun to them firing a shotgun.



*Screenshot from Broforce showing the character John Rambro*

While each “bro” in Broforce has their strengths and weaknesses, they are not all equally balanced as would be the case in the previously mentioned Brutal Legend. This can lead to competitive multiplayer games becoming incredibly unbalanced and effectively having the game favour how lucky you are, with regards to becoming a certain bro, rather than favouring player skill level. This was one thing that I definitely wanted to avoid with regards to my own project as I felt that balance was of the utmost importance.





*Screenshots from Broforce showing the terrain destruction feature*

One other game which inspired me was “Brutal Legend”. Brutal Legend is a Hack and Slash/ Real Time Strategy game. It is also however, unlike the previous two games, heavily inspired by metal music, just like my own project. This game involves the player playing as a roadie named “Eddie Riggs” who is transported to an alternate dimension where metal music itself has immense power. In the game you must unity various metal themed characters, most of whom are voiced by popular metal musicians such as Ozzy Osbourne (Black Sabbath), Lemmy Kilmeister (Motorhead) and Rob Halford (Judas Priest). With these characters united, you must defeat the death metal themed lord of evil, Emperor Doviculus by staging a series of concerts. These concerts involve creating a number of units of different types in order to destroy the enemies units and their stage.



*The world of Brutal Legend*

The variety of units allowed for a rock-paper-scissors-esque mechanic where each unit would excel against a specific enemy unit type while also being susceptible to a different enemy unit type. As previously mentioned with regards to Broforce, balance is of the utmost importance in this game and the rock-paper-scissors mechanics of Brutal Legend therefore are a definite inspiration for this project, however this will obviously need to be adapted to the weapons in the game. In addition to its unit balancing, one other aspect from Brutal Legend that I feel was impactful to my own game was the metal music theme of it. Outside of the metal musicians that voice characters in the game, the character moves and design, the story and world design are all heavily based around metal music. For example, the basic infantry unit for the traditional heavy metal faction in the game are called “Headbangers” and are essentially made up of a number of unarmoured units that are designed to head-bang objects and other units until they are destroyed. This is one aspect that I tried to include into my own project where units and the world itself were based around metal music and culture. 

*The player character running with “Headbanger” units in Brutal Legend*

### 2.3 Technologies Examined

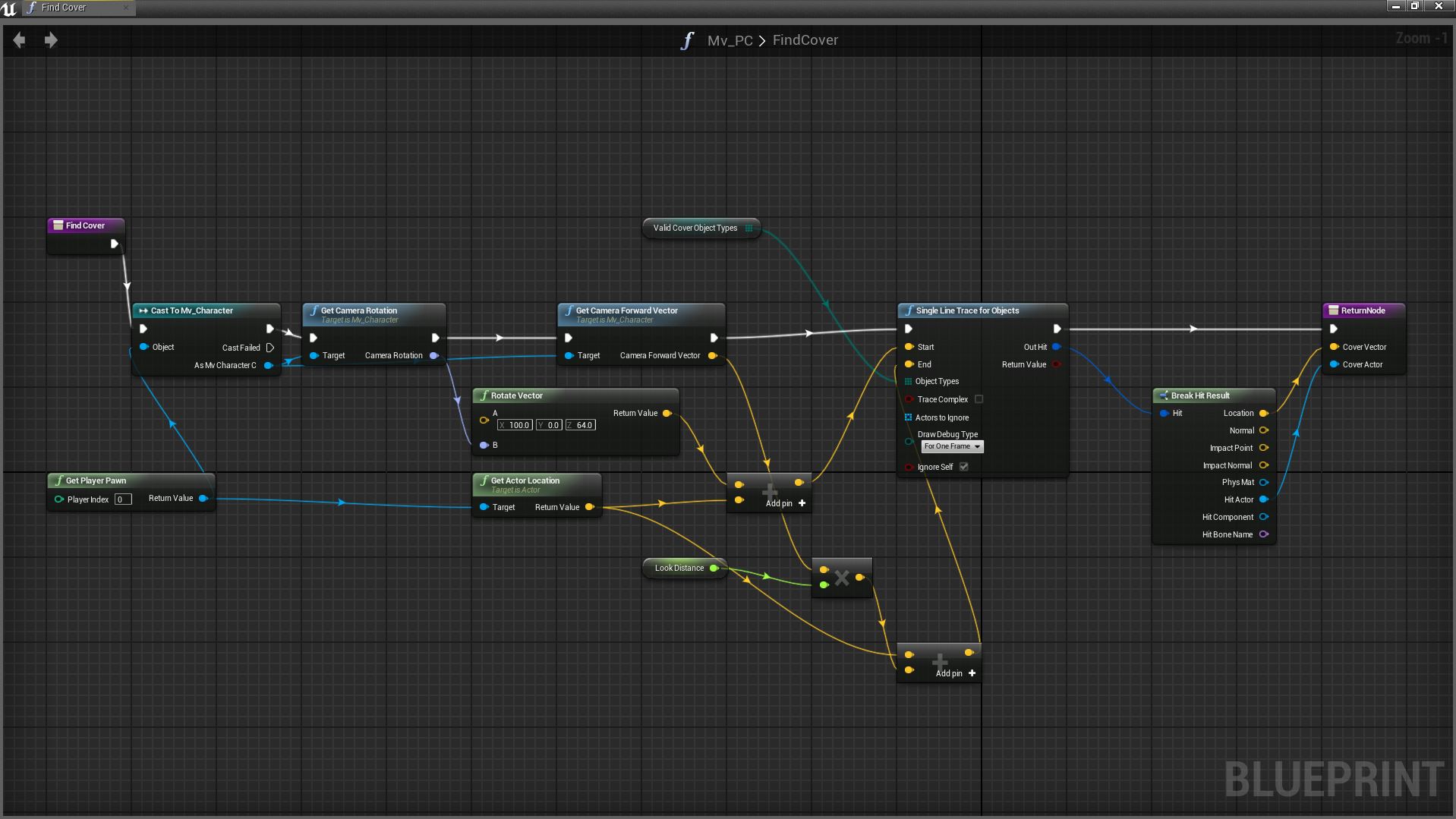
- Unreal Engine

- Cocos2D

- Unity Engine

As part of this project, I researched numerous game engines in order to determine which engine would be best for me to develop this project. The first game engine that I examined was the “Unreal” engine. The Unreal engine has been used to develop numerous high scale and budget games in the last decade or so. Some of these game include Bioshock, Gears of War 3 [unreal1 lp] and Fortnite Battle Royale. These are just a few of the note-worthy AAA games that have been built on this engine. One of the main reasons to utilize the Unreal engine over other game engines is that it is capable of producing very high fidelity visuals for games quite easily. It is often regarded as being capable of producing some of the best graphics in AAA games [unreal2 lp]. This engine however isn’t restricted to large scale development. The Unreal engine has been utilized by small scale independent studios such as the Terry Gilliam inspired “Rock of Ages”, developed by ACE Team [unreal3 lp]. However, the Unreal engine is more commonly associated with large scale AAA games.

The Unreal sports a number of advantages that can influence a developer to choose it over another rival engine. One of these advantages is that the complete engine is free to developers, this is however on the condition that games income revenue is less than $12’000 a year. The Unreal engine also offers multi-platform support for various systems, including but not limited to; Windows, Mac, Linux, PS4, Xbox One, IOS and Android. One of the biggest advantages of choosing Unreal over other engines is its “Blueprint” system. Unreal’s Blueprint system is a visual scripting system where users can create a series of interlinked nodes. When creating these nodes, code does not technically have to be written directly, this can allow for less adept programmers to more easily design games and more adept programmers to rapidly build prototypes.

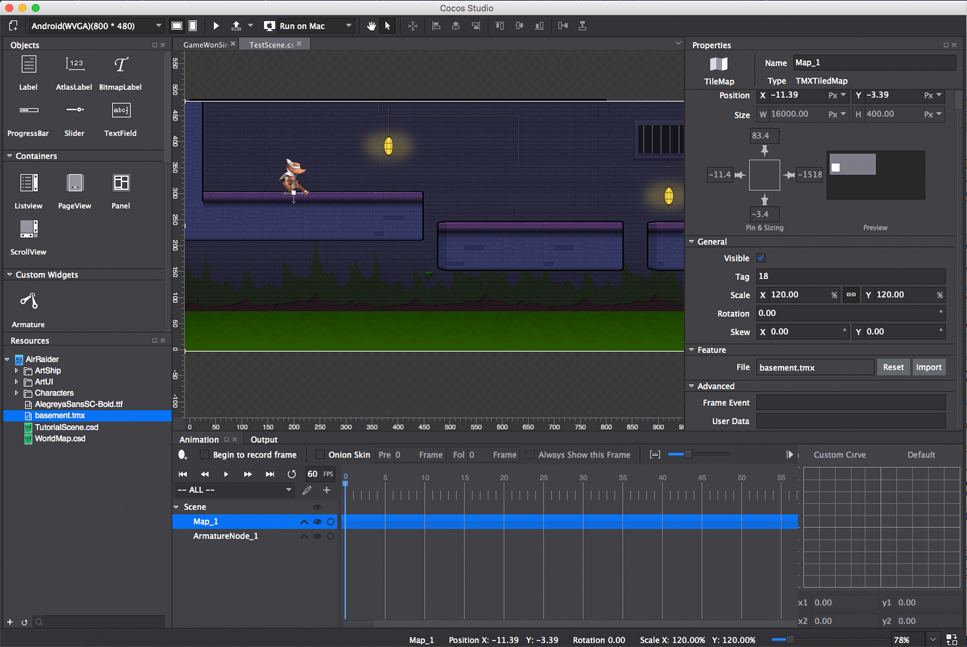


*The Unreal Engines Blueprint system*

There are however disadvantages to using this game engine over others. One of these disadvantages is that there is steep learning curve with the Unreal engine. Additionally, while Unreal does offer a number of tools to help in the creation of games using the engine, these tools are not particularly user friendly. Combining the lack of a user-friendly interface for these components with the already steep learning curve of the engine can result in the engine being incredibly difficult for developers to pick up [unreal4 lp]. Additionally, while it isn’t specifically a disadvantage, the Unreal engine is strictly C++ based. C++ is not a language that I am very familiar with and have had very little experience in developing applications of any type in it. This was one of the main reasons that I decided against using this engine however the previously mentioned steep learning curve also played a factor in my decision.

Aside from the Unreal engine, I decided to look specifically at engines that directly supported 2D game development as I had planned from the beginning that this project would be a 2D game. One of the engines that I discovered in my research was “Cocos2D”. While Cocos2D is generally not used by any AAA developers, it does have a loyal enough following of developers who decided to use it as their primary tool for game design. A survey of game developers found that 9% of them claim that this engine is primarily used for their work. While 9% of a group is generally not a large percentage on its own, when you consider the fact that this is when Cocos2D is up against large mainstream engines such as Unreal, Unity and CryEngine, 9% is a fairly respectable share to be able to claim [Cocos2D1 lp]. While Cocos2D isn’t normally adopted by large scale AAA publishers, it does have a decent foothold in the mobile games market. Cocos2D has been used by the company “Zynga” to develop some of its most popular games such as Farmville and Words With Friends [Cocos2D2 lp]. One of the biggest benefits of using Cocos2D as an engine is that it is completely free. Additionally, it is also open source. Due to the fact that this engine is open source, there are a wide number of user made tools and libraries that are available for the engine [Cocos2D3 lp] Cocos2D also supports multiple different languages such as Objective-C, C++, C# and Java.

There are however problems with using Cocos2D. For one, Cocos2D’s most widely supported platforms for development are mobile platforms and web based platforms. This was one of the main reasons why I decided against using Cocos2D as a game engine as I wanted this project to be playable at the very least on a computer that may not have access to an internet connection. Another problem that affects Cocos2D is that its API’s have been referred to as “unorthodox”. The reason being for this is that the engine was originally developed using Python. Part way through development however it was changed to Objective-C. Soon after that, it changed again and stuck with C++. During these transitions, some of the Objective-C expressions were maintained [Cocos2D4 lp].



*Screenshot of the Cocos2D User Interface*

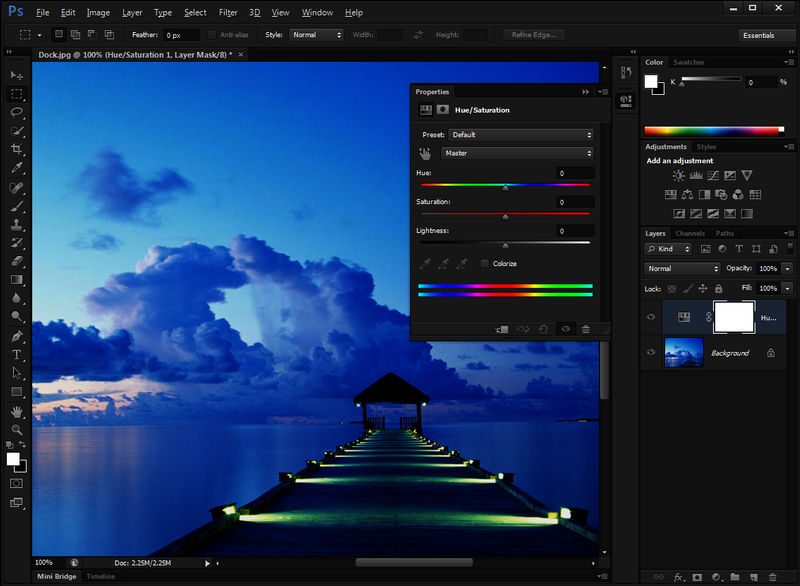
The game engine I inevitably decided to use was the “Unity” game engine. The Unity game engine is more similar to the Unreal engine than the Cocos2D engine in a number of different ways. One example of this is that Unity, like Unreal engine, is used in a number of large scale AAA games. There is however a very large independent game development community involved with the Unity engine too, more-so than the Unreal engine. Some of the games developed on this engine include; “Kerbal Space Program”, “Wasteland 2” and “Rust” [unity1 lp]. In the same survey that was mentioned while discussing Cocos2D, Unity was found to be used as the primary game development tool for 29% of developers. While Cocos2D specifically supports 2D, Unity supports development for both 2D and 3D games alike. Unity supports two main languages as far as scripting is concerned; Javascript and C#. Additionally, Unity also supports cross platform development tools allowing for development on all major console brands, mobile platforms and alternate gadgets such as smart TV’s. It is important to note that Unity also has a much gentler learning curve when compared to Unreal. The free licence for Unity covers all of the premium versions core features, the only major differences between the free and premium versions is that the premium version offers world creation tools, customizable splash screens and more detailed analytics. None of these things are particularly key or integral to this project so when deciding between the different versions of Unity to use, I chose to the free version.

Outside of the game engine, I also needed to decide on what software I would use to create the games art. As the graphics in this game were planned to be relatively simplistic, I decided to initially check out the default Microsoft Paint tool [image1 lp]. I decided against using Microsoft Paint quite quickly. The primary reason is that the tool does not natively offer transparent image types. This means that any image saved in this tool would have a large white box around the image.



*Screenshot from early build showing the difference between saving the same file in MS Paint (Left) and Adobe Photoshop (Right)*

I then decided to try to test out “Adobe Photoshop”, specifically Adobe Photoshop CS6 [image2 ]. This was due to the fact that I had experience in using the tool from using it for a number of years. Additionally, Adobe Photoshop offers a “Grid” feature. This places a grid around groups/individual pixels, which is extremely useful when attempting to create pixel art. Another reason why I considered using Photoshop over other applications is that it offers a layers system. This means that individual elements of an image can be separated onto separate layers, ensuring that changes on one layer, doesn’t affect any other layers. I originally planned to use the layers feature to store individual animation frames. I then planned to export these layers individually in order to generate character/object animations in the game.



*Adobe Photoshop CS6 Interface*

While the character in the picture above was designed in Photoshop, I felt that Photoshop was making certain things a little more awkward than they needed to be. I instead decided to search for programs designed for pixel art. One of the programs that I found was “Piskel” [image3 <https://www.piskelapp.com/> ]. Piskel ended up being exactly what I was looking for. It was a simple straight forward application with everything that I needed in order to create the pixel art I was going for. The features included covered everything from pixel dithering to live previews of the animation. Another key feature it offered is that piskel projects can have their frames exported as individual PNG files or in groups which would allow for easier animation implementation in Unity. It was for these reasons that I decided on using Piskel over other alternatives.



*Piskel Interface*

### 2.4 Alternate Research Done

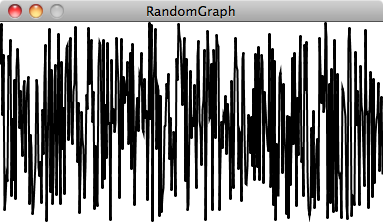
As a further part of my research for this project, I read the book “Unity in Action” by “Joseph Hocking” [book1]. Unity in Action is a book which covers many basic Unity functions for novice game developers. It contained multiple step by step guides, including pictures and code samples where appropriate. While this book does primarily focus more on the 3D aspect of Unity, generally, a lot of Unity’s 3D features can be converted to their 2D equivalent by replacing a 3 with a 2. For example, a 3 dimensional vector object is defined as “Vector3 vectorName(x,y,z)” while a 2D vector object is defined as “Vector2 vectorName(x,y). This book helped familiarise me with concepts and functions in Unity that I wasn’t originally familiar with.

Another book that I read was “The Art of Game Design – A Book of Lenses” by Jesse Schell [book2 lp]. While the previous book, Unity in Action, focused mainly on raw Unity functionality and code, this book focuses more on game design principles and how to best approach designing a game. One of the key principles that the book focuses on is the “Loop of Interaction” with regards to game interfaces. The principle itself revolves around the idea of how players and games constantly give each other feedback and the quality of this feedback can dramatically influence how much a player understands and enjoys a game. This is one principle in particular that I tried to take on board when designing this game.

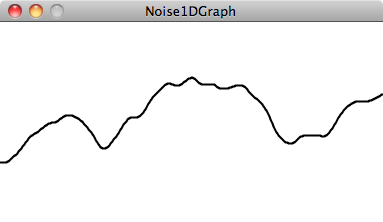
Another area I had to research briefly as part of this project was metal music as a whole and the stereotypes that come from each sub-genre and the bands that make up that sub-genre. For this, I decided to examine 8 popular sub-genres of metal that would each have their own unique characteristics. The characters design, name and nationality were all based around their particular sub-genre. Below is a table showing the bands examined as part of each sub-genre.

|  |  |
| --- | --- |
| **Genre** | **Bands** |
| Glam/Hair Metal | 1. Motley Crue 2. Twisted Sister 3. Kiss |
| Pirate Metal | 1. Alestorm 2. Lagerstein |
| Thrash Metal | 1. Slayer 2. Metallica 3. Anthrax |
| Death Metal | 1. Dethklok 2. Nekrogoblikon |
| Power Metal | 1. Gloryhammer 2. Rhapsody |
| Old-School British Metal | 1. Iron Maiden 2. Judas Priest 3. Motorhead 4. Black Sabbath |
| Alt Metal | 1. CKY 2. Linkin Park |
| Metal Mascots | 1. GWAR 2. Iron Maiden 3. Distrubted |

One final part of this project I had to research was what procedural generation algorithm I was going to use to build my game levels. The main source of my research for this section of my project was the “Procedural Content Generation Wiki” [procgen1]. The Procedural Content Generation Wiki is a knowledge base for all things procedural generation and includes lists of games that offer procedural generation mechanics. The site also has a section devoted solely to procedural terrain generation and all the algorithms that are commonly used by it. From this site, I was able to learn multiple different ways of how I could implement procedural generation in my game in terms of terrain generation. One of the algorithms I found while on this site was the Perlin Noise algorithm. Perlin Noise is a method of generating random numbers where the progession between values is dramatically smoother when compared to pure randomness. Below you can see a comparison between traditional random number generation and Perlin Noise random generation.



*Traditional random number generation*



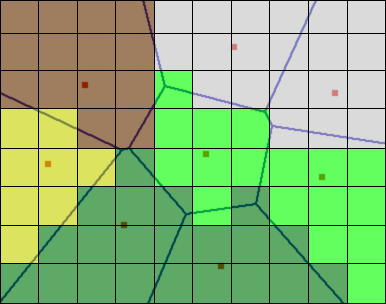
*Perlin Noise generation*

Since its development in the 1980’s, Perlin Noise has found itself used for procedural generation in a number of different fields, including gaming. One particular game which has used Perlin Noise for its terrain generation is Minecraft, which uses a hybridization of 3-Dimensional Perlin Noise and Whitaker Diagrams to create massive and diverse landscapes.



*An example of terrain generation in Minecraft*

While using Perlin Noise can result in extremely high quality areas, I felt that it wouldn’t exactly be suitable for what I wanted in terms of my own terrain generation. This led to me researching Voronoi Diagrams instead. This is an algorithm where a number of random points are taken from a map or grid. Neighbouring points/grid cells which have this Voronoi point as their nearest point are assigned to that point. One of the main reasons I had for examining Voronoi Diagrams in the first place is that they can produce more organic looking division compared to other algorithms and it is not nearly as complex as a Perlin Noise method of procedural generation may be. In my particular case, I only needed to use Voronoi Points of two particular types; Land and Air. As I felt that the simplicity of the algorithm as a whole would benefit the overall speed of the games terrain generation, I decided to use the Voronoi Diagram system.



*Voronoi Diagram example*

# 3. Design

### 3.1 Design Methodology

For this project, I needed to choose one development methodology that I would need to attempt to use throughout. As previously mentioned in the report with regards to “The Art of Game Design”, I wanted to place great emphasis on the “Loop of Interaction”. While in the book, it mainly referred to a player interacting with the game and the game responding appropriately, I wanted to try and adopt that style of interaction in my development cycle for the project where I constantly test the project (either from user testing or just general feedback) and adapt the project as needed. It was for this reason that I decided to use the agile life cycle, Scrum. I felt that by using Scrum, I could design the project in stages and get feedback from people at regular intervals, allowing me to change the project as needed as it was developed. By actively seeking feedback, I was able to avoid a situation where I could have developed and implemented a number of features in their entirety without getting actual feedback from a user who may not understand, or even like, some of the features.

Each iteration of the scrum methodology is called a “Sprint”. Each of these sprints are made up of a list of tasks that need to be done and implemented within a certain time frame. In order to successfully use this development cycle, I needed to prioritize certain features over others, which may be reliant pre-requisite features.

The features, and their level of priority, can be seen below.

1. Core Mechanics:
   1. Player Movement
   2. Weapon Pickup and Drop Down Functionality
   3. Basic Weapon Firing Mechanics
2. Initial Spritework (Single frame of animation per character for testing)
3. Basic Procedural Generation of Worlds
4. Base Design Menu User Interfaces
5. Terrain Destruction
6. Dynamic Terrain Generation
   1. Item Spawning Locations
   2. Explosive Bomb Box/Spike Pit Generation
   3. Power-up Spawning
7. Character Animations
8. Asynchronous Terrain Generation
9. Dynamic In-Game Camera
   1. Dynamic Positioning
   2. Dynamic Zooming
10. Different firing mechanics per weapon
11. Finalizing UI
12. Final User Testing
13. Seed Generation

While the core complexity of this project is its procedural generation of terrain, I felt that it would be more important to prioritize the core mechanics of the game over everything else. My sprints for this project were based primarily around this model. Extra time was allocated for longer or more difficult tasks such as Character Animations and Dynamic Terrain Generation.

### 3.2 Project Component Design

1. Design
   1. Identification of a design methodology including why it was chosen
   2. Design of each of the project components eg: the UI, Network, Project Demonstration, source code layout
   3. Clearly identifying the list of features and use cases supported within the project.
2. Architecture & Development
   1. Overview of the system architecture and a diagram to represent all of the key elements within the architecture.
   2. Details of each component within the project, problems encountered and resolved, challenges overcome or worked around.
   3. Identify key development components;
   4. Identification/explanation of external APIs used versus own code ; List of classes of your code etc .
3. System Validation
   1. Testing
      1. What testing was performed, why it was selected and what are the key use cases within the project.
   2. Demonstration
      1. Identify what features can be demonstrated and show screen shots or reference a video online to show the project demonstration (for audience not at demo)
4. Project Plan
   1. Project Plan analysis and review of how it changed from the initial proposal including explanation of what changed and why, and suggestions on how to address this if the project was repeated.
5. Conclusion
   1. Analysis of the projects key elements identify the key learning obtained from the project and recommendations and suggestions for how the work can be improved on continued into the future.
6. Bibliography
7. Appendix