Hemmo Latvala

Designing and Developing a Mobile Game

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Technology

Bachelor's Thesis

9 April 2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Hemmo Latvala Mobiilipelin suunnittelu ja kehitys 34 sivua + 6 liitettä 9.4.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Toimitusjohtaja Aki Kolehmainen Lehtori Antti Laiho

Insinöörityössä pyrittiin kehittämään ominaisuuksiltaan valmis mobiilipelin prototyyppi. Tavoitteena oli saada peli kehitettyä vaiheeseen, josta sen saisi vähällä työllä julkaisukelpoiseksi. Lopullinen peli tulee olemaan paljon suorituskykyisempi ja monipuolisempi.

Mobiilipelit eroavat konsoli- ja tietokonepeleistä monella tapaa. Matkapuhelin pelialustana tarjoaa uniikkeja mahdollisuuksia, mutta myös rajaa pois ratkaisumalleja, jotka toimivat moitteitta muilla alustoilla. Tämän takia kehittäjän täytyy tarvittaessa kyetä poikkeamaan tutuista käyttöliittymäkonventioista.

Pelin kehityksessä pyrittiin tarkastelemaan käytetyn pelinkehitysympäristön tarjoamia ominaisuuksia. Kehitysprosessi oli iteratiivinen ja sovelsi Lean Startup -metodologiaa, jotta peli saisi mahdollisimman paljon testausta ja palautetta alusta lähtien. Jokaisella iteraatiolla peliä pyrittiin saamaan helpommin lähestyttäväksi ja viihdyttävämmäksi pelaajalle.

Pelin laadun ja viihdearvon varmistamiseksi työssä painottuivat erityisesti käyttäjäpalaute ja iteratiivinen kehittäminen. Puolessa välissä kehitystä peli ei saanut hetkeen palautetta, ja tämä näkyi kehityksen laadussa. Loppua kohti käyttötestaamista jälleen hyödynnettiin, ja se sai aikaan erinomaisia lopputuloksia.

Insinöörityön tulos oli prototyyppi, joka innosti pelaajia pyrkimään ennätysten rikkomiseen ja antamaan ideoita pelin syventämiseen pienillä lisäominaisuuksilla. Prototyypin olennaiset pelimekaniikat voitiin todeta menestyneiksi, ja projektia aiotaan kehittää edelleen palautteen pohjalta.

Avainsanat	pelikehitys, Unity

Author Title Number of Pages Date	Hemmo Latvala Designing and Developing a Mobile Game 34 pages + 6 appendices 8 April 2015
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Aki Kolehmainen, CEO Antti Laiho, Senior Lecturer

The aim of the final year project was to produce a prototype mobile game. The objective was to develop a prototype that would be quick to finalize into a releasable product. As such the game shall be far more performant and diverse once released.

During development the features provided by the Unity game development environment were examined. The development process was iterative and it implemented the Lean Startup methodology in order to provide the game with sufficient testing and feedback from the early stages. The game was to be made more easily approachable and entertaining for the player with every iteration.

The importance of user feedback and iterative development are emphasized in the thesis as they proved crucial in ensuring the game's quality and entertainment value. Half-way in development, no user feedback was collected for a brief period of time and this became evident in the quality of work. Towards the end, user testing was once more employed with excellent results.

The product of the project was a prototype, which inspired players to strive for breaking past highscores and to provide ideas to deepen the game with minor additional features. The game's core game mechanics could thus be deemed successful. As a result, the game will be developed further based on feedback.

Keywords	mobile games, Unity

Table of Contents

Appendices

Appendix 1. Evolution of Player.cs movement functionality

Appendix 2. Examples of Shark.cs

Appendix 3. Examples of CollectibleSpawn.cs

Appendix 4. GUIButton.cs

Appendix 5. Scoring and Game Over behaviour

Appendix 6. Parts of LevelGenerator.cs

Abbreviations

2D	Two-dimensional. In game terms it often refers to an object that uses two dimensions, the X and Y vectors.
UI	User Interface
SDK	Source Development Kit

1 Introduction

Mobile games are now the most monetarily lucrative subset in the game industry. As of the first quarter of 2014, there are 200 companies in the Finnish game development industry, which are largely mobile focused. Over half of the companies were founded only a short while ago. It is estimated that they employ over 2,400 people. The total turnover of the Finnish game industry in 2013 reached 900 million euros and vastly eclipsed the industry's turnover from 2012, which was a mere 250 million [1]. Mobile game development is more cost efficient and simple in comparison to modern major titles for consoles and the PC. This is largely true as the core audience consists of a relatively new demographic of casual players, who are after a more light-hearted gaming experience. Thanks to this the games do not necessarily need much depth to be worth the small price to pay to play them. Mobile games also require less development time in general due to often being much smaller in scope than games on other platforms. In addition, the target audience for mobile games is more all-encompassing as a large portion of the world's population owns a phone capable of running games.

Syntax Eror Poroductions (commonly referred to as Syntax Eror) is a company that was formed with the goal of developing mobile games. The company was founded by Petteri and Aki Kolehmainen alongside six other IT professionals in 2013. I joined the company last year. So far the company has developed and published one game, Subconscious. The first game took a long time to develop and launch due to a heavy focus on story and level development. As a result the next games to be developed aim to go for a more casual approach. The games will need to take less time to develop and the game needs to be able to be played only for 5 minutes at a time or even less.

In the fall of 2014, the company held a brainstorming phase where developers would present their ideas to each other and the most engaging ideas were chosen to be developed further. I provided the small prototype concept of a game that got named Get 2 the Chopper. The goal of the game is simple. The player pilots a helicopter and uses it to rescue people from danger. This scenario provided the possibility of developing the gameplay for brief gaming sessions. The company's goal for 2015 was releasing a few games within the year and seeing what works. Get 2 the Chopper was

chosen to be one of the games to be developed as a result of the brainstorming session.

A primary goal of Syntax Eror is to produce quality products. For this to be possible, the games development process needs to be a learning process. The aim of this study is to discuss current practices and tools of mobile game design and development through producing a casual mobile game for the iPhone and Android platforms. Furthermore, the goal of this study is to establish what kind of tools and a mind-set are required for mobile game development.

2 Project Background

There are significant differences in how developers approach designing a mobile game compared to console and computer game development. Due to the phone and tablet being easily portable, the core target audience tends to play mobile games as a brief means to pass time instead of dedicating hours of consecutive gameplay. For comparison the average gaming session on a phone or tablet lasts roughly a little over a minute while on games in other platforms the average session lasts over an hour. [2;3] This puts different pressures on the mobile game developer as the user base they target is much faster to lose interest and more prone to playing a short amount of time overall.

A recent study indicates that a mobile game intended for casual markets needs to convince the player within five minutes of starting the application that the game is worth playing. As a result the game needs to be introduced and taught in that five minute time frame or the players with the shortest attention spans drop out and never truly dedicate time to the game [4]. This is a crucial factor as most games are distributed as free-to-play and rely on in-game ads and micro-transactions for revenue [5].

Game development has many options at the very start of development. Choosing the right tools and development processes is crucial. Understanding the core nature and requirements of the project is crucial, so the developer can pick the right approaches for their projects. In addition to mastering the right tools, a designer will need to understand the current landscape as well as the history of gaming as a whole to properly produce a game that will make use of the lessons learned from past developers and the success stories of the present.

This chapter will introduce the tools and landscape of game development as well as the goals for the Get 2 the Chopper project. This includes the state of the industry, potential customers, popular development tools and practices.

2.1 Project Requirements

2.1.1 Casual Gamers as the Target Audience

When gamers are discussed, the focus is often wrongly on the relatively niche demographic referred to as hardcore gamers. These players are often very invested in the games they play. Gaming sessions often tend to last multiple hours and the games that cater to this audience also reflect this in their gameplay often resulting in very intricate mechanics that encourage the player to immerse themselves fully to the game world for the absolute best experience. As recent studies have shown, the hardcore demographic is being eclipsed by the casual gamers. [5;6]

By comparison to the hardcore, the casual gamer is a completely different kind of player. A casual gamer is not so easily inclined to dedicate hours upon hours of time on a single product, nor is he or she eager to spend as much money on acquiring the game initially. This is not to be mistaken as the casual and hardcore being totally different people. A primarily hardcore-focused gamer may enjoy playing the occasional round of Angry Birds, Bejeweled Blitz or 2048 on their work commute to pass the time, much like a casual gamer may engage in playing a few rounds of hardcore games in social gatherings with friends or by themselves, but just not in equally large quantities as a more hardcore player. [6;7;8]

Many people think of games such as Farmville and Clash of Clans as casual games. While in nature they are the very epitome of casual gaming, it still does not mean that a casual gamer solely plays games like Farmville. Any game can be played casually. A casual gamer simply plays far more leisurely and for shorter periods of time when compared to a hardcore gamer, who will spend hours on playing. [9]

To create a casual game, the developer does not have to borrow mechanics from slow-paced and leisurely games. When designing the core features that make up the experience, the features need to be tailored to fit into brief sessions of game play. This allows the player to hop on and off at will despite what is going on in the game and suffers no real setbacks.

Many casual games are played on mobile devices. Immersion is clearly far weaker due to the game being played on a smaller screen, often outside of the comfort of the

player's own home with sound effects disabled, since that might bother surrounding people. The lack of several senses being stimulated at once keeps the player very grounded in reality.

For the casual gamer, a game acts as a time sink – a chance to change focus from the regular tasks to something entertaining. The key features that win this kind of a gamer over are art, replay-ability, price and familiarity. [11.]

2.1.2 Cross-platform Compatibility

Syntax Eror develops games with the idea of developing both iOS and Android versions. This means that the Get 2 the Chopper game needs to be developed on a game engine that provides the possibility of distributing the game for multiple platforms. The company's development team has largely used Unity as their engine of choice for their projects.

In the modern game development environment, cross-platform development is very easy. Developing non-game applications for multiple platforms is far more challenging as the developer will have to take the varied user interface (UI) conventions of different devices into account. [12] It does not require much effort for the game developer to be able to launch the project on the platform of their choice. On the Unity engine, for instance, the developers merely need to have access to their chosen platform's source development kit (SDK) and they will be able to compile the project. [13] This is why Syntax Eror prefers Unity.

Valid competitors to Unity on providing this support for the developer include:

- Corona
- Cocos2D JS
- Unreal Engine 4
- Appcelerator Titanium

3 Tools and Practices

A project requires a clear development cycle to keep it on track and the right tools to produce good quality as fast as possible. This chapter discusses the background and nature of the tools and practices that were chosen for the Get 2 the Chopper project.

Software development has an abundance of development models. This project was developed with an iterative approach as this allows for swift improvements and changes when needed without requiring long term dedication to a single plan. Figure 1 illustrates the work flow of the project. Each iteration started with ideas, which were then produced in the Unity game engine using the MonoDevelop integrated development environment to program the game's behaviour in C#. The outcome was then stored in GitHub and tested on Android and other mobile devices. Once the iteration received some feedback, a new cycle began with fresh ideas based on the outcome of the previous iteration. Lean start-up is the development model employed.

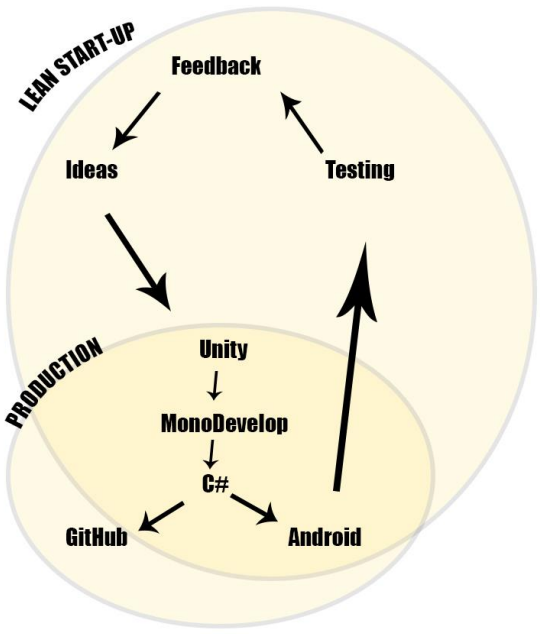

Figure 1. Workflow of the project.

3.1 Unity

With the requirements of fast development and the necessity of being able to launch on multiple platforms, the tools used needed to be very nimble and fast to iterate on. For this reason Unity was chosen as the development environment.

Developed by Unity Technologies, Unity is a game development ecosystem. It provides a wide range of crucial features. A rendering engine with fully integrated tools and workflows to create interactive 2D and 3D content as well as multiplatform publishing and the Unity Asset Store ensure a developer hardly needs to leave the Unity environment while working. [14.] Like its competitors, it features a WYSIWYG (what you see is what you get) editor. This makes work exceptionally streamlined as the developer will not have to go in and out of the game to check if everything works as intended when developing.

In the past game development started with creating the game engine. This required the developer to spend a lot of time creating basic mechanics, sound and graphics renderers. In contrast modern game development often relies on engines that provide WYSIWYG editors.

Today, over 45% of game developers favour Unity as their game engine and development environment [14]. Like the majority of mobile game developers, Syntax Eror also uses Unity. Prior to my involvement with the company, I had already done several projects on Unity, but none had focused on mobile games. In the latest releases Unity has provided very useful tools for mobile game development, which are ideal for my project.

Unity was originally made solely for 3D game development, but as of version 4.3, Unity provides a 2D game viewport, which makes setting up games that only use two dimensional objects far more simple. In addition there's now a set of pre-made behaviour scripts geared for 2D games. [15] They provide ready-made physics, camera and controller behaviour. This means the developer can ideally skip any game engine behaviour programming and start developing gameplay.

A game developed on Unity can currently be released on Windows, Mac, iOS, Android, Blackberry, Windows Phone, Linux, PlayStation 3, PlayStation 4, PlayStation Vita,

Xbox One, Xbox 360, WiiU as well as the Internet Explorer, Safari, Chrome and Firefox browsers. The version that was used for this project (4.6) is not able to compile the games for browsers that do not have the Unity Web Player installed. However, as of the newly arrived Unity 5, it is possible to build WebGL applications, which run on any modern browser capable of running on hardware that allows for graphics acceleration in-browser [16].

This flexibility in release platforms allows any game developed on Unity to be launched on every major mobile phone operation system without much need for tedious code ports, but launching on any platform is possible through small adjustments in the build configurations. Syntax Eror aims to launch their games at least on iPhone and Android application stores.

Unity has several key competitors, some of which are only capable of providing a portion of what Unity can provide. The Corona SDK is a large competitor for the mobile game developer audience. It provides largely the same services as Unity with a key difference being the supported languages. Corona SDK relies on Lua as its language of choice [17].

While Corona is a strong competitor in mobile game development, the Unreal Development Kit, developed by Epic Games competes with Unity in every aspect. Featuring a robust engine and state-of-the-art development tools, UDK has become popular among triple A standard game studios. For a long time, it provided many more powerful tools and graphics compared to Unity. A large difference between Unity and UDK was the monthly subscription fee, which changed as of March 2015. UDK used to require a monthly subscription from the developer. Then Epic Games announced that the entire Unreal Engine is available for free [18]. Shortly afterwards Unity announced that it would provide the entire functionality of Unity 5 for free for anyone to use [19]. The competition between development environments is very vigorous.

3.2 MonoDevelop

Unity provides a customized version of MonoDevelop as an integrated development environment, often abbreviated as IDE. An IDE is a text editor which provides additional features useful for programming such as debugging. MonoDevelop is

primarily designed for C# and other .NET languages, but it is possible use any language on it. The Core is available under the LGPLv2 license, but a large portion of the code and add-ins are licensed on the MIT/X11 license. The entire source is available online. [20.]

MonoDevelop comes with a wide array of key features, which include the following:

- Linux, Windows and Mac OS X support.
- Code completion support for C# as well as code templates and code folding.
- Language support for C#, F#, Visual Basic .Net, C, C++ and Vala.
- An integrated debugger for Mono and native applications.
- Tools for Source control, makefile integration, unit testing, packaging and deployment and localization.

[20]

3.3 C#

C# is a programming language which was initially conceived in the late 90s and early 2000s in the midst of the early development of .NET by Microsoft. Its development was led by Anders Hejlsberg. It was meant as an alternative to other object-oriented programming languages, such as C++ and Java, which were at the time the most popular. [21.]

The programming language is intended to be simple, modern, general-purpose and object-oriented. Software robustness, durability and programmer productivity were emphasized in its development. One of the language's goals is to provide support for software engineering principles which include strong type checking, array bounds checking, detection of attempts to use uninitialized variables and automatic garbage collection [22].

In Unity, the developer is given access to several programming languages to develop with. C# was chosen for the Get 2 the Chopper project as Syntax Eror's other projects are also developed with it. This means that the features created during the project can easily be applied to others as well.

3.4 GitHub

Syntax Eror utilizes GitHub for storing all their projects and keeping everyone's code up to date. Founded by Tom Preston-Werner, Chris Wanstrath and PJ Hyett in 2007, as of 2015, GitHub hosts over 20.6 million code repositories and provides everything necessary for version control in public and private projects. In addition to providing a code repository, it provides additional features such as issue tracking, team management and other online collaboration tools. [23.;24.]

3.5 Android

Originally developed by Android Incorporated, Android is a mobile operating system based on the Linux Kernel. It is mainly designed for touchscreen devices such as smartphones and tablet computers. Its early development was financially backed by Google. In 2005, Google bought the company and continued development in-house. [25.]

As figure 2 depicts, Android's market share has been on a steady climb and it has went on to become the biggest phone operating system.

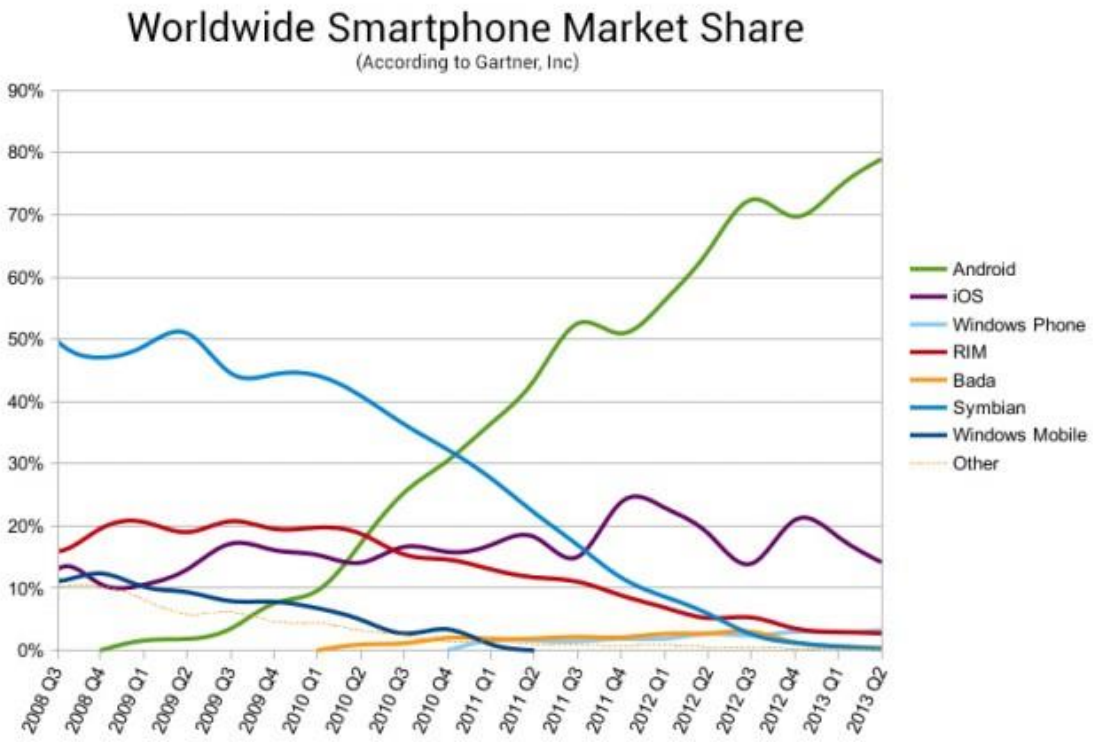

Figure 2. Worldwide Smartphone market share through the years. [27]

Despite Android being open source, its codebase is strongly controlled by Google as forks of the operating system will not be supported by the company in any way or form on the Google Play store via updates [26].

As of 2013 the Google Play application store has become the largest of its kind. This is due to the increase in popularity amongst developers as well as share on the mobile phone market. [27.]

I utilized an LG G2 smart phone which runs on Android 4.4 as the primary test machine for this project. Colleagues from Syntax Eror aided in testing the game for the iOS devices.

3.6 Lean Start-up Development

Games never quite turn out as they were planned initially. What you first designed might not be fun and engaging, so further iterations are necessary to create an enjoyable experience. As a result, the most common philosophies for game

development are flexible methods. This means that the project was carried out with a very iterative approach.

The most integral part of the methods for this project was the iterative development process. The software was broken down into small components. In games this usually means the various gameplay features and behavior. Each feature is designed, produced and tested build by build. This means a feature is first designed thoroughly. Once the design is sound, it will be implemented as specified. Upon completion it will receive testing and assessment. Based on the results of the last phase, it will either be fully approved or put back into a new development iteration loop, which will use the feedback gained from the previous iteration as the basis for the follow-up design and implementation. [28.]

Lean start-up development focuses on getting the product out for testing as early and often as possible. This ensures that a bad idea will not be developed for long because of a risk of investing time and money on something that will eventually flop as the concept was just not practical from the start. [28.]

During this project I developed prototypes and early versions which were shared with friends and colleagues as often as possible to receive direct feedback to keep the project on track.

4 Initial Design and Prototyping

The goal of the project was to create a playable and feature-complete game. The game had to be simple to learn and it had to be engaging to play on a mobile device. The project was established with a simple concep: the game should replicate the gameplay experiences of Flappy Bird and Robot Unicorn and provide some of its own unique twists.

Flappy Bird was developed by Max McDonnell. The game has the player guide a bird through narrow gaps in walls by adjusting the bird's altitude by tapping the screen, which causes the bird to flap its wings. [29.] The game boasts very simple gameplay, which is easy to learn. I wanted an equally intuitive control method for my game. Despite being easy to control, Flappy Bird is very difficult due to every obstacle being equally hard to fly through. Flying through obstacles requires absolute focus and feel for the game to be successful. Roughly 98% of games played end in a score lower than 10 [30].

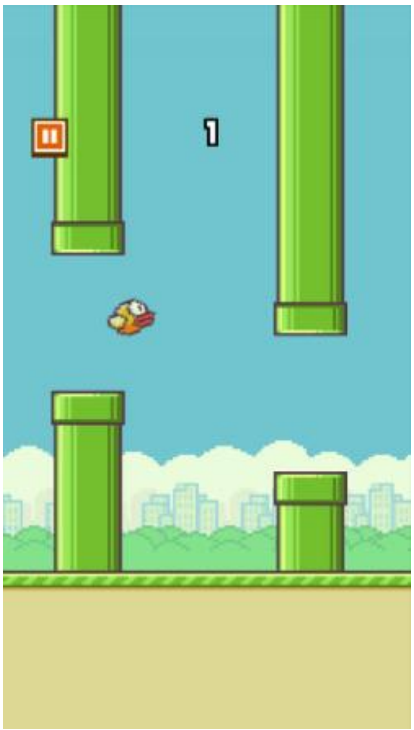

Figure 3. Screenshot of the Flappy Bird browser game. [30]

Developed by Spiritonin Media Games, Robot Unicorn Attack is a game where the player is a robot unicorn that gallops through a fantasy setting jumping over pitfalls,

collecting fairies and dashing through stars that block its path [31]. Similarly Get 2 the Chopper needed some goals and challenges beyond surviving past hurdles to add some complexity and variety. Robot Unicorn Attack's collecting gameplay serves as a good example of how to add more challenge to the game.

Figure 4. Screenshot of Robot Unicorn Attack [31]

After pitching the idea to Syntax Eror and they asked for a quick prototype. In a week, a small game was ready. It was reminiscent of Flappy Bird's level design. It had a helicopter which the player moves by touching anywhere in the screen surrounding it to give momentum in that direction. The helicopter could be flown around and it was able to collect people on-board. If the helicopter touched a wall, it would spin out of control and die.

Essentially it was a very crude fusion of Flappy Bird and Robot Unicorn Attack with my own control scheme. I had not given much thought on the game beyond just the general concept of flying a helicopter and getting people aboard it as I wanted to keep the project as barebones as possible to be able to shift direction based on initial feedback.

By the end, the prototype had the following features:

- The helicopter can be moved by holding a finger on the screen.
- The helicopter can die by hitting a wall.

- The helicopter can collect people.
- An arbitrary score tally goes up whenever a person is collected

After showcasing the prototype to the rest of the Syntax Eror team, a brief brainstorming session was held where the game received its first feedback and the go-ahead to carry on with the development. The feedback was as follows:

- The helicopter was too hard to pilot.
- The game needed a clear goal
- It would be cool if the people who were picked up affected gameplay

The prototype was very crude and had no elements that made it an actual game. Based on the early feedback, the follow-up plan was to place the game's story into a rescue scenario, where people were saved from an accident site.

Work began with changing the control scheme and behaviour. Instead of tapping on a position to make the helicopter move there, a more conventional solution of a small touchpad was chosen. The touchpad could be used for applying momentum in the direction of the player's choice. Implementing it was simple, as Unity comes with a ready-made touchpad prefab. The advantage of this over the previous was that the user now does not move his/her finger around the screen to move around in the game. Instead he/she can keep his/her control finger roughly in the same small area and tap a specific are of the box to apply momentum.

Unity's Rigidbody2D component was used to apply gravity and mass to the helicopter game object. It provides all the necessary methods and behaviour that a physics object would require to interact with Unity's physics engine [32]. However, the problem with the controls was that the player had to keep tapping constantly to apply more momentum, because the helicopter was being dragged down by gravity. The movement did not feel very helicopter-like either. Gravity was disabled for the helicopter to address this issue. This change made the movement feel much more appropriate and in control of the player.

Figure 5 explains the difference from the user interface standpoint and also serves to demonstrate what the first two versions of the game were like.

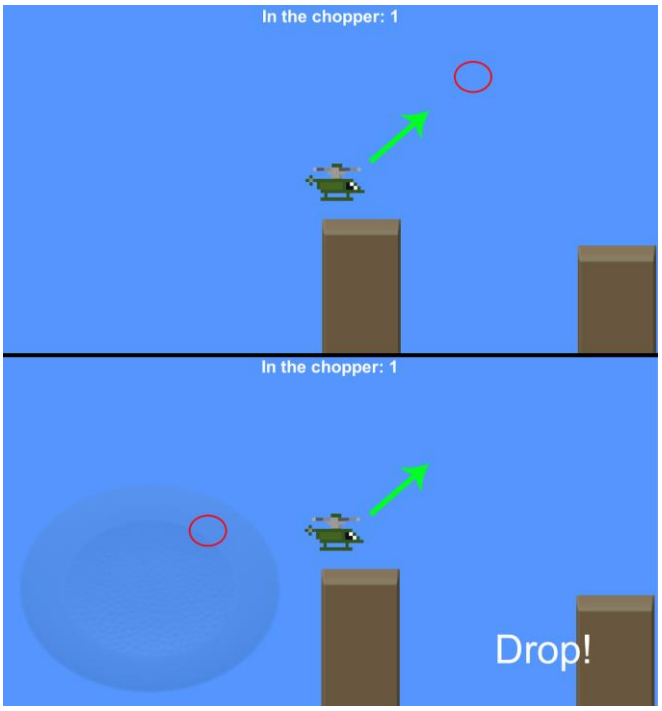

Figure 5. Screenshot of Get 2 The Chopper movement logic variations.

The red circle indicates the point where the user is touching on the screen and the green arrow indicates the direction in which momentum is applied for the helicopter. The first image illustrates the first prototype while the second shows the newer iteration. The second image also features a button for dropping survivors who are aboard the helicopter. This is explained later in this chapter.

The first excerpt from player.cs in appendix 1 demonstrates code from the prototype that initially handled movement within the helicopter object's Update function, which is called once per rendered game frame. What this did was it fetched the position of the mouse – or in this case the finger – and gave it a positive or negative value based on the location it was in within the monitor. These values were then multiplied by 200 which served as the speed modifier to apply the ideal amount of momentum. The X and Y forces were then stored to a Vector2 variable, because the Rigidbody2D component's AddForce method accepts only Vector2 variables [32]. This method then applied the momentum to the object. With this the player was able to apply movement for the duration he/she kept touching the screen.

Due to the aforementioned issues with forcing the player to touch all-over the screen to move, the code was improved upon by replacing the screen touch position calculations with values that Unity's own Joystick script returns when touching the touchpad [32]. This solution can be seen in the second excerpt in appendix 1. Additionally, a public float variable called speed was introduced as a modifier, which could then be used to more easily adjust the movement speed based on testing in the future.

After improving movement, focus shifted to working on the goal of the game. As discussed during the brainstorming, the helicopter could be used for rescuing people, so a rescue zone was introduced. This is a spot where the player can drop off the people he has collected.

I came up with the concept of the rescued players holding onto the helicopter instead of simply being seemingly devoured inside the helicopter. This served to address the issue of the collected people not really affecting gameplay. Now they were holding onto the helicopter forming a chain of people as the player collected more of them. An example of the hanging behaviour as well as the rescue zone is depicted in the figure 6.

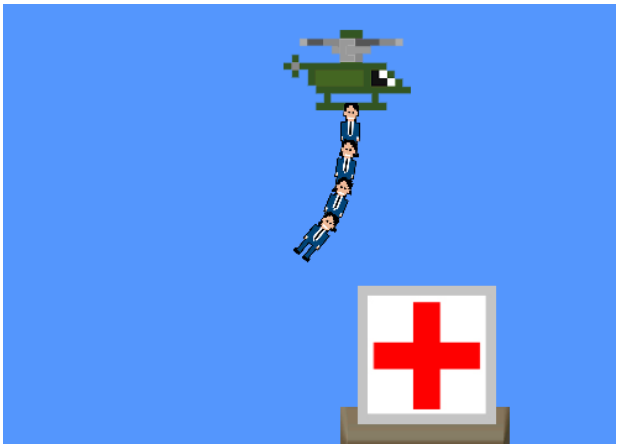

Figure 6. Carrying people in Get 2 The Chopper.

The chain of people was created by using the Hingejoint2D component provided by Unity, which allows an object to attach itself to another object that uses Rigidbody2D as well. To keep people from behaving with physics before being collected, the component is initially set as disabled on the GameObject, but it is then enabled upon

collection event. HingeJoint2D provides a variable called Connected Body to which the desired connected object's Rigidbody2D has to be assigned. [32]

The Collected method has as a constructor which can be used to forward the target GameObject to the method. The game object provided for method is decided in the collectible survivor's OnTriggerEnter2D function where upon entering the hitbox of the player or another survivor will make some simple game logic checks and decipher what game object it should be connected to.

A lot of time was spent working on the logic that keeps track of the people hanging onto the helicopter. After trying out different types of arrays available in Unity, C#'s List class was chosen as it allowed for the necessary flexibility that was needed for adding and removing objects from the index with the Remove method. The method allows for simply passing the game object that needs to be removed from the array, as a variable. The List class creates a list of objects that can be accessed by index. In addition to the Remove method, List also features methods for various types of searches, sorting and manipulation. [33] The system collection array which was available in Unity appeared far weaker in terms of finding specific items from the array. To employ this array type instead of List would have required creating new array handlers, which would have taken a great deal of development time.

After removing the collected item, the code also checked if the object was the first person connected to the helicopter with a Boolean variable that returns true if that is valid. Then the helicopter will now know that it does not have any people aboard anymore. This was very important in making it possible for the helicopter to be able to collect people correctly. This boolean was used to prevent the helicopter from picking up more people. Once a single person is holding onto the helicopter, the player has to collect people by making them touch the ones already hanging on.

5 Development

After the early features of the prototype were finished, another meeting was held with Syntax Eror. As before I showcased the game to the rest of the staff and they provided insight in to the game. The overall vibe was positive, and people felt that the way people were collected was very fun which added a lot to the gameplay experience. There was discussion of adding enemies and different kinds of people to rescue, but no clear plans were set for the future.

The meeting ended up being the last one for multiple months. This gave a lot of time to work with minimal contact to the rest of Syntax Eror. Most of this time was spent in refining and polishing the existing features.

5.1 Features

Core gameplay and features were largely established at the prototyping stage. The game lacked features that would make it stand out as a finished product. The art in the prototype consisted entirely of placeholders so it had to be designed and created from scratch. The game had no additional perils for the player beyond challenging controls, so it required additional depth and value. There was no clear end game where the player would know the game was over either.

This chapter document the thought processes and actual development of these features.

5.1.1 Art Style

For a long time the game was developed with pure placeholders that represented rough estimations of what the objects could look like. As the game was nearing completion in terms of features, it was time to take a deeper look into the art of the game.

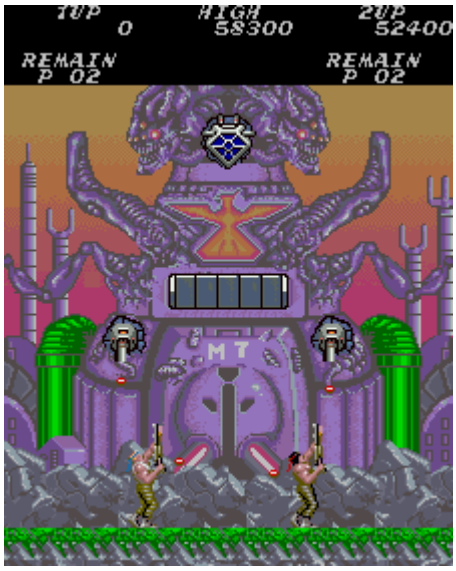

Figure 7. Screenshot of *Contra* [34].

As a starting point, the goal was set to draw heavy inspiration from old games. The initial idea was to have pixel art similar to Contra (see figure 7) and other 1990's console games. The idea slowly moved away from that concept as it would require sprite-based animation and without a dedicated sprite artist, there would simply not be sufficient time to create animations that would be satisfactory. Despite this, it was thought that a retro vibe could still work. The concept simply evolved towards a look and feel that was much older – Tennis for Two (see figure 8).

Figure 8. Tennis for Two [35].

Tennis for Two was one of the first video games ever developed. It was built on radar equipment by William Higinbotham in 1958. What I found very attractive with it was the reliance on a single colour and the apparent fluorescent glow around the game objects. This had a very simple aesthetically pleasing element to it. [35]

Combining this with the idea of other monochrome video game classics such as Pong, which is pictured in figure 9, I set out to draw simplistic glowing silhouettes of the objects to be featured in the game.

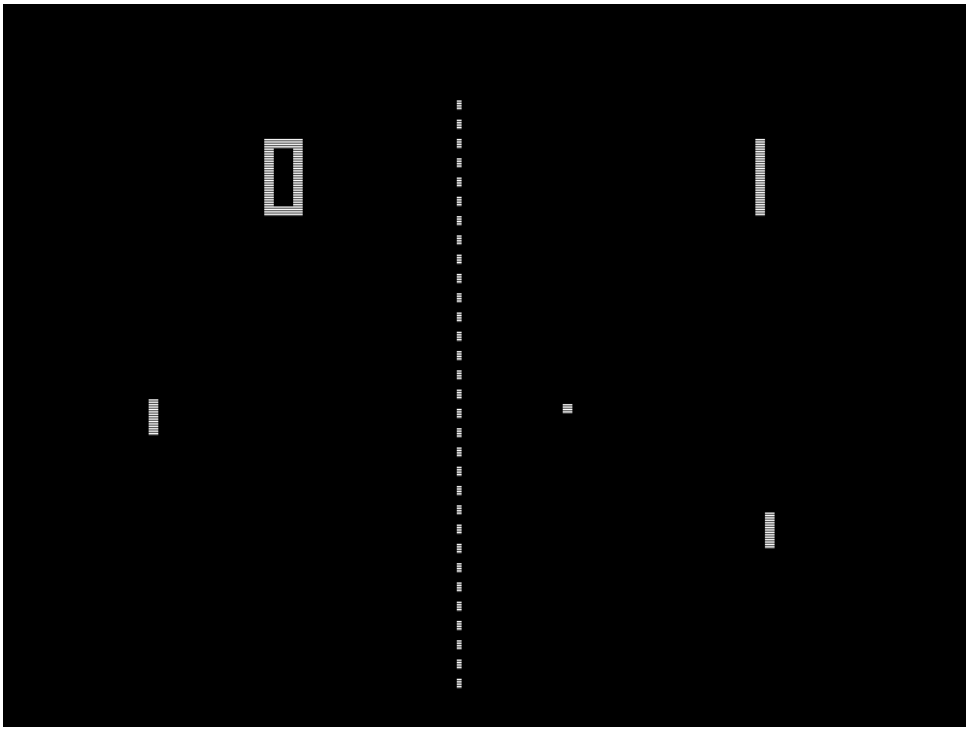

Figure 9. Screenshot of Pong.

Combining the glow from Tennis for Two and the general black and white vibe resulted in the first two characters for Get 2 the Chopper that were put together using Adobe Photoshop CS6. The characters were drawn in vectors to ensure they can be made in any resolution in the future. To make them animated, the characters were made in pieces. Figure 10 below depicts the sprite files for the helicopter and human.

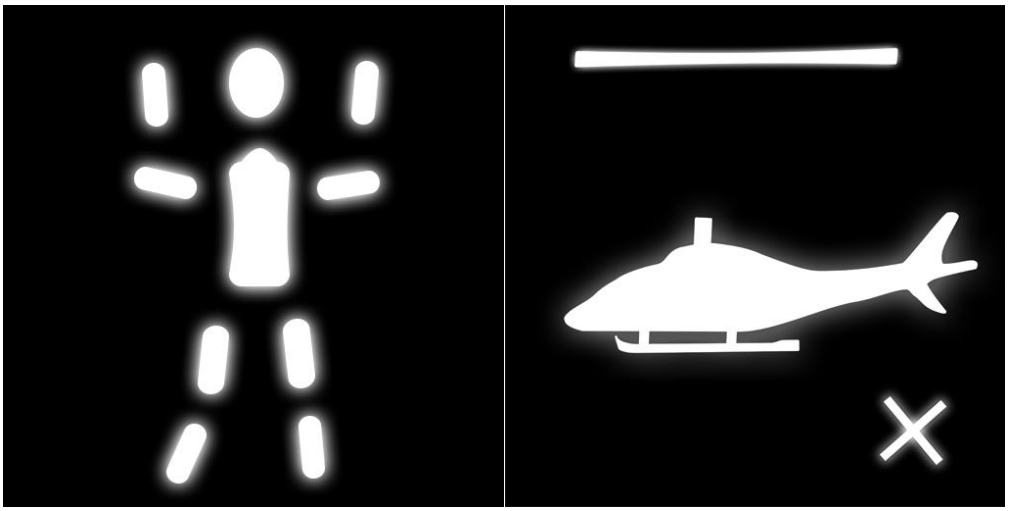

Figure 10. Sprite sheets for the human and helicopter characters.

To animate the characters, Unity's Animator component was used. Unity also provides a newer component called Animation, which would have allowed IK rigging of the human character. The older component was used as it felt like overkill to use the powerful and more complex Animation component. The characters would be using very simple animations, so developing any more complex behaviour would have been largely pointless. [32]

Animating the human character was simple. Each limb was connected to its respective part in the anatomy. In general, Unity animation allows the objects to be locked from one position, which will then become the pivot point when the object is rotated. Once this was done for the character, the animations were created with the Animation view, where each limb's motion was keyed individually to create the desired movements. The animations created for the human were:

- Frantic splashing to stay afloat in the water
- Holding onto the helicopter
- Struggling to hold onto the helicopter

The human animations get called in the code through methods provided by the Animator component. Animator allows for crossfading between animations, which allows for smooth transitions from one animation state to another. The methods are very flexible as the programmer can specify the duration of the transition between the two animations when calling the Crossfade method. [32]

The other objects in the game were far simpler to animate as they consisted of only a single piece or have some objects attached that simply needed to be rotated, such as the helicopter rotors.

5.1.2 Enemies

The game also needed threats and danger to keep it fresh. For this reason, a simple enemy character was created that follows the player around and performs attacks when close. The game's scenario was chosen to be a shipwreck, so the most logical enemies were sharks. Considering the very retro feel of the game, it somehow made sense to have sharks trying to attack the helicopter.

The shark character needed to be able to move towards the helicopter on the X axis only. To know in which direction the character needs to be going, a function was created that gets the distance between the player and the shark.

As seen in the CheckTarget excerpt in Appendix 2, the code compares the target position on the X axis with the enemy's position by reducing the player's position from the target's position and then returns the difference. This is called in the MoveTowards method, which gets called in the shark's Update method.

The returned value will then be used in the movement method. Movement direction of the shark is assigned to be left or right based on whether the value is negative or not. In addition to using the value for direction checks, it is used for distance checks to help decide when to attack. In the third excerpt in Appendix 2, the same method is used to check if the player is within a certain range of the shark. If that is the true, then it will set the jump Boolean to true and start applying the jumping logic instead of normal movement logic.

5.1.3 People and Spawning

Since the idea was to create an infinite level, the people that the player needs to rescue had to be spawned infinitely. For this purpose a spawner needed to be implemented. The spawner is essentially a co-routine that gets triggered on round start or whenever the person that the spawner previously instantiated is picked up or

drowned. The function that calls for the spawner simply applies a random timer for the next spawn method.

A co-routine is like any other function, but the key difference is that it can be put on hold without disrupting the flow of other functions. A normal function has to happen within the single game frame it was called in, but a co-routine runs on a separate thread until it has been performed. This allows for creating wait timers by calling the yield function. The function has to return an IEnumerator to work. Appendix 3 provides an example of starting a co-routine in the TriggerSpawnEvent method. [32]

The Spawn co-routine started by the method waits for the amount of time given with the yield function and then it performs two rolls with Random.Range to decide what type of a person to spawn as well as where to spawn him. It also initiates the collectible game object variable as it will be assigned to different game objects depending on the spawn roll. Once a person is spawned, he will be sent a message telling that this spawner is responsible for tracking him. This is used so that upon collection or death, the person's script can inform the spawner that it is free to begin spawning again.

To add some challenge and urgency to rescuing people, two other coroutines were added. One for drowning the people if they are not picked up from the water in time, the other to make them able to hold on to the helicopter for only a limited amount of time. This way the person would have to be picked up fast or he would drown and be lost. If a person has been holding onto the helicopter for too long, he will fall off. The implementation is very much the same as for the spawner with one difference. The drowning co-routine is stopped, if the person is picked up by the helicopter. This is achieved in much the same way as starting a co-routine by using the StopCoroutine method. The method requires a string value that matches an active coroutine's name and then terminates the targeted routine. [32]

5.1.4 Controls

Even though the controls seemed to worked, I often found the helicopter having too little power when moving in its designated direction. It was also hard to move precisely. Playing the game on the keyboard felt very nice and intuitive. Having 4 directions that the player could go towards and then be able to press up and right arrows to move top-

right was very nice. This level of control would also be necessary for the mobile version to ensure responsive and enjoyable gameplay.

Anna Marsh, developer of Lady Shotgun Games, said in an interview for Gamasutra, *"I sit down with the device, and start imagining how the touch actions should work. I used to do this with a controller too, when I worked on console -- sit there pretending to play the game pressing the buttons.*" [36] Her way of pretending to be playing gave a lot of food for thought on how to set out to change the game's controls for the better. I picked up the test phone and moved around my imaginary helicopter.

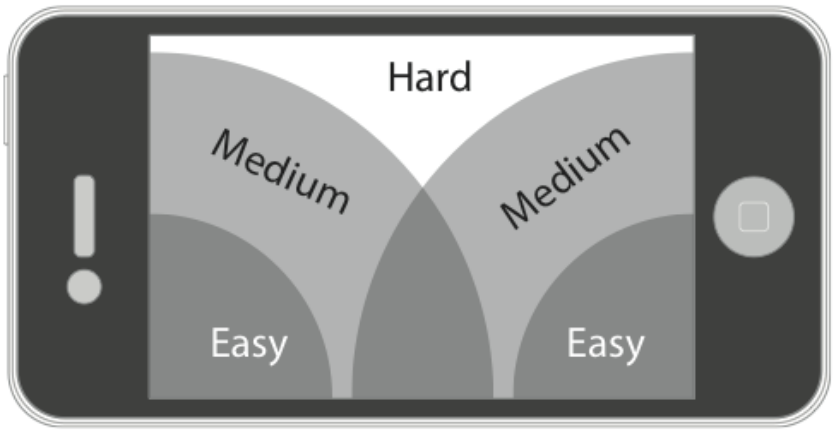

Figure 11. Illustrating accessibility of different regions on a mobile [37].

When setting out to design the new control scheme, I imagined how I would use my left thumb to move up and down, while using the right thumb for left and right movement. It proved very easy to interact with buttons placed in the bottom right and left corners this way. Adam Telfer points to the same conclusion in his article Touch Control Design: Ways of Playing on Mobile. As figure 11 shows, the areas where the user has the easiest time of reaching with thumb-based controls are situated in the bottom left and right as well as bottom centre. [37]

What I came up with was having both thumbs used for controls. As figure 12 illustrates, opaque buttons were placed for each movement direction on the corners of the screen. For example, this allowed the user to press both up and right at the same time by using two fingers. The bottom centre button was reserved for the drop button.

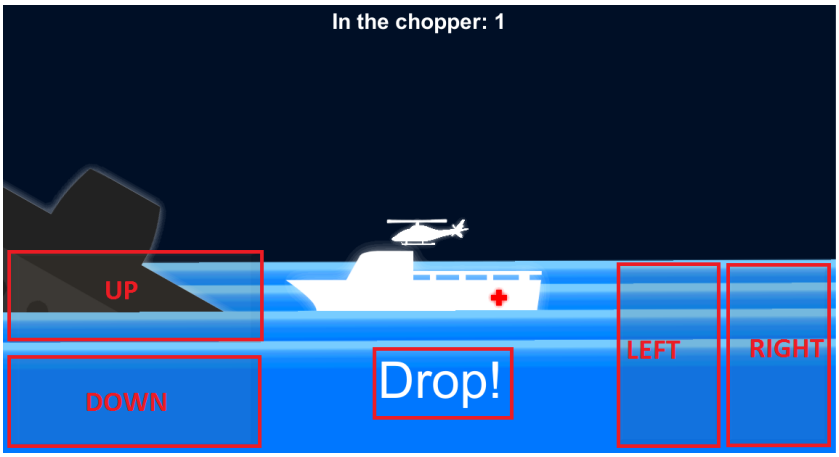

Figure 12. Controls for Get 2 the Chopper.

After playing around with the new button layout for a few hours, it felt like the perfect solution. All the UI buttons were given a simple script, which communicates the commands to the player.

This script can be found in Appendix 4. The button upon initiation calls the Start method that finds the player game object, which will then be accessed by the two methods: OnMouseDown and OnMouseDrag. The former is called upon the event of a mouse clicking the game object that the script is attached to. The latter differs in that it is called as long as the mouse button is held down and it is over the game object. These methods can also be used for touchscreen interfaces. [32]

The script has two public variables which are used to create buttons with different kinds of behaviour. The string variable called Command is to be set as the name of the Player script's method that should be called in the script. The mouseDown Boolean allows selecting whether it allows the button to be held down or not.

The script can easily be repurposed for any similar needs later. It uses the slightly less performant SendMessage method, which should not be used for scenes with a lot of things going on at once. For a simple mobile game SendMessage should not cause any performance issues. It is best suited for UI features such as menu buttons. [32]

5.1.5 Scoring and the End Game

The game also needed an arbitrary method of grading the player's performance. For this a score counter was introduced. The Score script attached to it receives messages from the GameControl script, which works to keep track of the game state as well as the points scored by the player. Appendix 5 features excerpts of the methods that communicate with each other.

Upon dropping a person to the rescue zone, the rescued person sends a message to the game controller to run the ScoreUp function with the score value of that specific person. This then proceeds to update the UI game object by calling its SetScore function.

When scoring was all set, the next step was to make a simple end game screen. The figure 13 illustrates what the end result looked like. The entire end game UI is wrapped inside a Unity prefab, which is spawned in the GameControl script upon the dying Player script sending the "Game Over" message to it.

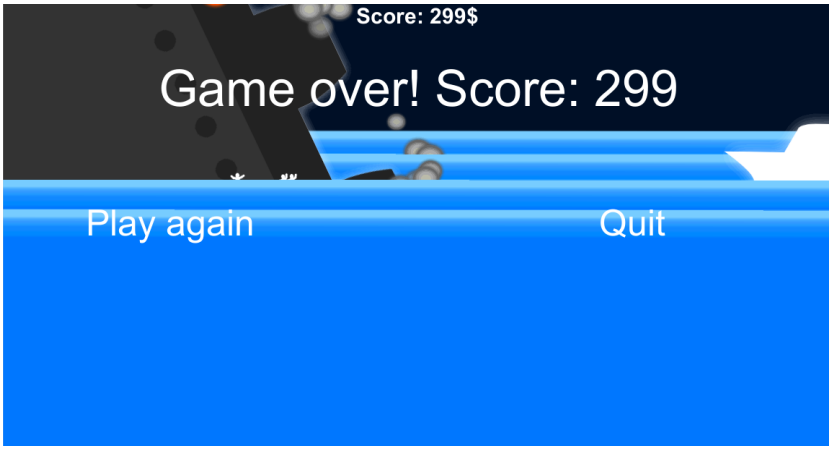

Figure 13. Get 2 the Chopper's game over screen.

The GameOver method first sets the gameOver Boolean as true and then destroys the gameplay controls. Then it instantiates the Game Over UI from the prefab resources and assigns it to the gameOverButtons variable. The Game Over UI prefab has a game object called winText. The code finds the object with the name and then calls its script and sets its text based on current score. Once all this is done, the code also disables all the spawners so no more people keep spawning in the game over screen.

5.2 Alpha Phase

When feature development had largely reached an end in late February and the game was very much in a playable form with everything essential that was planned for it, the game was ready for the early alpha stage. Figure 14 illustrates the game at this stage. To add some visual flair, I had included a sinking ship to further establish the game scenario for the player.

The game featured four enemy sharks with randomized movement speeds, five survivor spawners and a rescue zone for the people. The game would end only upon the helicopter being destroyed by sharks or flying too deep into the water.

Figure 14. Gameplay shot of Get 2 the Chopper.

5.2.1 Testing and Feedback

After optimizing the existing features the game was shared with 8 testers. The game was presented to them as it was. To examine the ease of learning the game, no guidance was given. The testers had to figure out movement in the game on their own. This was done to make sure that the controls were intuitive. Controlling the player character is the most crucial part of the success of a mobile game. It needs to be quick to learn and it needs to feel like the player is in perfect control.

Besides focusing on the intuitivity of the controls, I collected some general feelings from the testers as well.

The following observations were made based on watching the players:

- Players kept swiping the buttons.
- Players tried dodging objects that were meant to be in the background.
- Players felt like the game did not have much purpose.
- Players felt lost in the level.

After asking for an honest answer from one of the testers, he said, "*I don't think people will play this more than once. There's just not enough challenge and the game feels very slow-paced.*" This feedback alone made me think whether the game had strayed too far from the original concepts set up for the project. The user interface had five buttons and the game did not increase in difficulty as it progressed.

It became very clear that the game was too complicated to be a mobile game meant to be played casually. The player would need to be heavily involved to first master the controls and then the actual game itself.

5.2.2 Re-design

After receiving the testing feedback it was decided that drastic changes were needed for the game. It had to be simplified. While getting caught up in the minor details of the game, the big picture and initial goals had been lost. This game was meant to be similar to Flappy Bird and Robot Unicorn Attack and it had strayed very far from its target gameplay.

When evaluating Get 2 the Chopper against LeBlanc's taxonomy of game pleasures [38], it seemed to be lacking. It provided a minor narrative in the form of saving people from the shipwreck and some slight challenge as the player had to evade sharks while saving people. Neither of these pleasures was fully realized.

To figure out the next step of development, the two games responsible for most of the inspiration were examined. This was a way of mapping what kind of pleasure they first and foremost provide the player.

The two games had four aspects in common.

- Infinite sidescrolling.
- Distance travelled increasing the player's score.
- Evading the level's terrain to get further.
- One mistake meant game over.

This indicates that the two games primarily derived their pleasures from challenge and masochism. They urge the players to beat their previous high scores and to do so despite the game's high difficulty. Combined with competing against other people's scores, the games get even more pleasurable. These four features are very crucial to causing pleasure for the player and they could easily be better designed for this game as well.

5.2.3 Improvements

The following changes were to be made:

- Removing left and right directional control completely
- The helicopter moving right at gradually increasing speed
- Adding a level generator that builds new obstacles and events as the player progresses.
- Creating a user interface that does not require any visible buttons.

The first two changes were simple to implement as they simply required removal of the controls needed for them. Forcing the helicopter to move right on every frame was just a matter of calling upon the MoveRight function on the Update function.

To simplify controls further, all the buttons were removed. Some of the old code for screen tap position based movement was reintroduced and adjusted to only apply movement up or down based on whether the screen was touched on the top half or the

bottom half. Instead of applying the force directly as in the first prototype, this version called the Up and Down functions which were created for the buttons to access previously. Unlike in the early prototype revisions 1 and 2 in Appendix 1, where movement strength was applied based on distance from the helicopter, the new version allows the force applied to be consistent despite of where on the screen the player touches.

The level generator was a much bigger task. A complex script was created for it which was then attached to the game's main camera. All of the key events were handled in the script's Update function, which is called on each game frame to check for changes and player progress (see Appendix 6).

The code starts by storing the current position of the camera object that this script is attached to into a variable which it then uses to check if it has gone past the current interval. How this interval worked is that the script had an index variable, which starts off from 1. This index is multiplied with the interval variable. This was arbitrarily set to 70 as any game object placed within range of the screen centre would be out of the camera's field of view. Whenever the camera moves far enough right, the interval is reached and as a result the index is increased. Every single time the camera travels 70 more units to the right, the code will trigger again.

After this the script does a dice roll using the Random.Range method to give a value between 0 and 5. This is used to decide whether to spawn a rescue zone or a block that has a collectable person standing on top of it. These two methods place their designated items to the next interval.

In addition to placing the block to the next interval, the function also places it to a random height. This generates some variety and challenge to the game as the player will have to keep adjusting altitude to avoid crashing into the blocks.

Once a block is spawned, the code proceeds to check for the threat interval in the next if statement. This is logic that was added to make the code generate dangers at a predictable rate. The RollThreat method gets called on start-up before anything else is run. This assigns the threatInterval integer a value between the minimum and maximum interval size. This is then compared to the threatCountdown integer which counts the number of intervals it has passed since the last time there was a threat. One

interval before the threat is generated, the code calls for the Warning method, which creates a 3D text game object which flies across the screen to indicate that there is danger ahead. SpawnThreat is very similar to the SpawnBlock method. The key difference is that it spawns its designated game object at different heights.

Once these features were set, the game was once more ready for testing. The gameplay had changed drastically and the forced movement really generated a far more engaging gaming experience.

5.2.4 Retrospective

After 9 months of work, the game reached its final stages of development before the big cut-off. However, after the last testing round I felt very confident about the game I had managed to create. It felt engaging and the gameplay was far more intuitive and did not have similar issues as the previous iteration. While the changes were drastic, this is what creating a proof of concept prototype is about. The figure 15 is a screenshot of the typical gameplay.

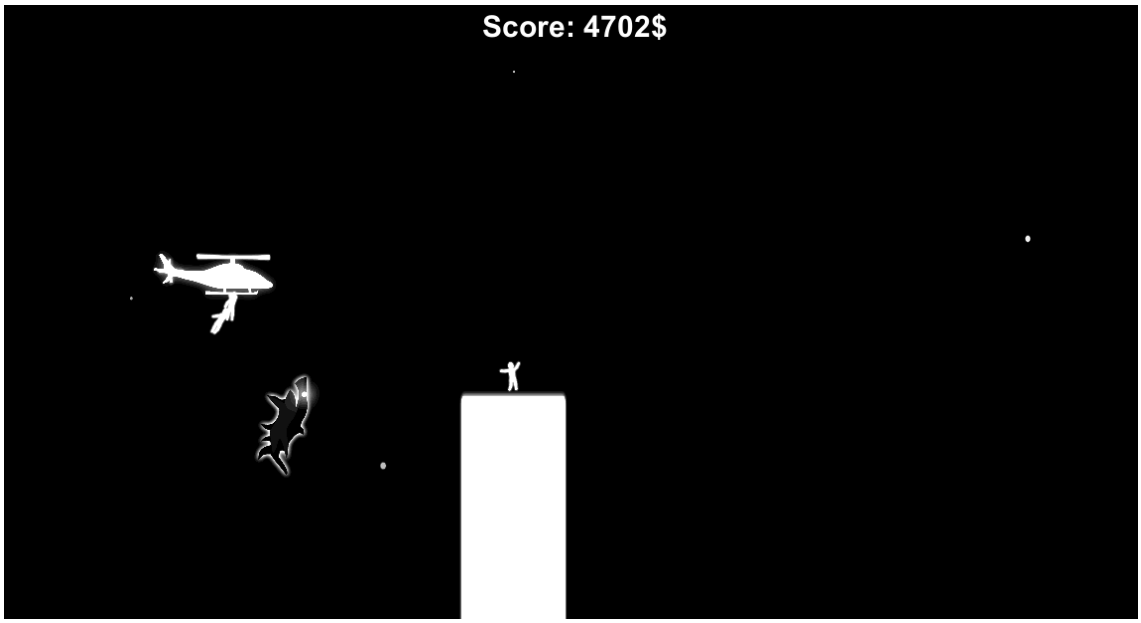

Figure 15. Gameplay screenshot of Get 2 the Chopper.

The new version was presented to the testers as well Aki Kolehmainen from Syntax Eror to receive their feedback. The reception was largely positive. Most stated that the

game actually felt like a game and that it had replay value. The issues with control confusion were abolished and players felt like they were progressing constantly.

In addition to positive feedback, the testers came up with several ideas for how to follow up and expand the game. This was good to hear as it showed how players were genuinely excited to play more since the game had a little more variety. Compared to the first test round, the feedback was much more excited. Everyone considered the game to be genuinely fun. The core gameplay was finally refined enough for the game prototype to be considered finished.

6 Conclusions and the Future

Designing a mobile game is vastly different from developing for the PC or console platforms. There are many things that cannot be achieved on the mobile platform, but there are also several that can only be done on it. While Get 2 the Chopper largely follows principles of classic game design, the user interface proved to be challenging to develop. On touchscreens, people enjoy touching and interacting with the game objects presented to them. Overcomplicating the controls and gameplay was an obvious detriment to the progression of the project. Had I initially gone with a more simplified game and discarded my idea of complex movement sooner, the game could have been far more complicated and feature rich in the same time frame.

The importance of playtesting the game early with friends and colleagues cannot be emphasized enough. Every single occasion I gave the game for someone else to try, I was met with enthusiasm, ideas, criticism and bugs. All of these are important for the early stages of a game's development as they help the developer in discovering what exactly makes their games tick.

The game is still not finished, so it will continue its development even after writing this thesis. The next features in the pipeline include:

- Picking up higher value people and objects.
- Different kinds of perils to evade including police helicopters and heat-seeking rockets.
- Flying too high will cause the helicopter to stall briefly as opposed to simply explode.
- A karma system to punish leaving people behind or getting them killed.
- Procedurally generated background graphics such as in-game ads in the form of billboards.

Unity provides a very powerful set of tools for the mobile game developer and it keeps expanding its repertoire with constant support and updates. During the making of this thesis, Unity updated from 4.6 to Unity 5. [19] There was never a time when I felt restricted by the engine. My own imagination was often more limiting than the tools available. In the future I will potentially seek to try out using the Unreal Engine for

mobile game development, but now I know for certain that Unity is there to provide a reliable engine and set of tools for a mobile games developer.

As someone who started off as a pessimist towards mobile games, the more I read and experienced them first-hand, I began to appreciate the amount of genuine planning and engineering it takes to create the perfect experience on a portable device. Creating a game is not about making a complex system, but a simple user experience with depth for those willing to experience it.

Despite straying far away from its inspiration sources initially, Get 2 the Chopper ended up very close to what had been initially planned – a fusion of Robot Unicorn Attack and Flappy Bird. It has the same sense of urgency and unforgiveness while still giving its own touch of gameplay, humour and aesthetics to stand out from its sources of inspiration.

References

1 Chapple Craig. Finland: The Games Start-up Capital of the World [online]. Develop; 28 April 2014. URL:http://www.develop-online.net/region-focus/finland-the-games-start-up-capital-of-the-world/0192199. Accessed 8 April 2015.

2 Böhmer Matthias, Hecht Brent, Schöning Johannes, Krüger Antonio, Bauer Gernot. Falling Asleep with Angry Birds, Facebook and Kindle – A Large Scale Study on Mobile Application Usage. Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services New York: ACM Press; 2011.

3 McMillan Gavin. Insights on Casual Games – Analysis of Casual Games for the PC. The Nielsen Company; 2009.

4 Järvinen Aki. First Five Minutes: How Tutorials Make or Break Your Social Game [online]. Gamasutra; 21 April 2010. URL: http://www.gamasutra.com/view/feature/132715/first_five_minutes_how_tutorials_.php. Accessed 8 April 2015.

5 Galarneau Lisa. 2014 Global Gaming Stats: Who's Playing What, and Why? [online]. Big Fish Games, Inc; 16 January 2014. URL: http://www.bigfishgames.com/blog/2014-global-gaming-stats-whos-playing-what-and-why. Accessed 8 April 2015.

6 Mason Mike. Demographic Breakdown of Casual, Mid-Core and Hard-Core Mobile Gamers [online]. Magmic Inc; 19 December 2013. URL: http://developers.magmic.com/demographic-breakdown-casual-mid-core-hard-core-mobile-gamers. Accessed 8 April 2015.

7 Angry Birds [online]. Rovio Entertainment Ltd; 2015. URL: https://www.angrybirds.com/play/angry-birds. Accessed 7 April 2015.

8 2048 [online]. GitHub, Inc; 2015. URL: https://gabrielecirulli.github.io/2048/. Accessed 7 April 2015.

9 Barefoot Heather. In Defense of the Casual Gamer [online]. Defy Media, LLC; 31 October 2013. URL: http://www.escapistmagazine.com/articles/view/video-games/editorials/10699-In-Defense-of-the-Casual-Gamer. Accessed 20 March 2015.

10 Chiapello Laureline. Formalizing Casual games: A Study Based on Game Designers' Professional Knowledge. DiGRA; 2013.

11 Rogers Alex. Launching and Marketing a Mobile Game: Strategy and Consumer Perceptions. Metropolia University of Applied Sciences; 2012.

12 Ristolainen Juha. Multi-Platform Mobile Development with Xamarin [online]. Futurice; 12 September 2014.
URL: http://futurice.com/blog/multi-platform-mobile-development-with-xamarin. Accessed 8 April 2015.

13 Weiss Nat. Selecting a Cross-platform Game Engine [online]. Binpress; 14 May 2014.
URL: http://www.binpress.com/blog/2014/05/14/selecting-cross-platform-game-engine/. Accessed 8 April 2015.

14 Unity [online]. Unity Technologies; 2015. URL: http://unity3d.com. Accessed 18 March 2015.

15 Goldstone, Will. Unity Native 2D Tools [online]. Unity Technologies; 28 August 2013.
URL: http://blogs.unity3d.com/2013/08/28/unity-native-2d-tools/. Accessed 8 April 2015.

16 Echterhoff Jonas. On the Future of Web Publishing in Unity. Unity Technologies; 29 April 2014.
URL: http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/. Accessed 8 April 2015.

17 Corona Labs [online]. Corona Labs; 2015.
URL: https://coronalabs.com/. Accessed 1 April 2015.

18 Sarkar Samit. Epic makes Unreal Engine 4 free [online]. Vox Media Inc; 2 March 2015. URL: https://www.polygon.com/2015/3/2/8134425/unreal-engine-4-free-epic-games. Accessed 15 March 2015.

19 Dingman Hayden. Unity 5's New Full-Featured Personal Edition is Completely, Utterly Free to Use. IDG Consumer & SMB; 3 March 2015.
URL: http://www.pcworld.com/article/2892314/unity-5s-new-full-featured-personal-edition-is-completely-utterly-free-to-use.html. Accessed 15 March 2015.

20 MonoDevelop [online]. MonoDevelop Project; 2015.
URL: http://www.monodevelop.com. Accessed 13 March 2015.

21 Hamilton Naomi. The A-Z of Programming Languages: C# [online]. IDG Communications; 1 October 2008.
URL: http://www.computerworld.com.au/article/261958/a-z_programming_languages_c_/. Accessed 14 March 2015.

22 ECMA-334 4th Edition: C# Language Specification. ECMA International; 2006.

23 Preston-Werner Tom. GitHub Turns One! [online]. GitHub, Inc; 19 October 2008. URL: https://github.com/blog/185-github-turns-one. Accessed 16 March 2015.

24 GitHub [online]. GitHub, Inc; 2015. URL: https://github.com. Accessed 16 March 2015.

25 Elgin Ben. Google Buys Android for Its Mobile Arsenal [online]. Bloomberg L.P; 17 August 2005. URL: http://www.webcitation.org/5wk7sIvVb. Accessed 17 March 2015.

26 Amadeo Ron. Google's Iron Grip on Android: Controlling Open Source by any Means Necessary [online]. Condé Nast; 21 October 2013. URL: http://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/. Accessed 15 March 2015.

27 Victor H. Android's Google Play Beats App Store with over 1 Million Apps, Now Officially Largest [online]. phoneArena.com; 24 July 2013. URL: http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680. Accessed 13 March 2015.

28 Ries Eric. The Lean Startup Methodology [online]. Crown Business; 2015. URL: http://theleanstartup.com/principles. Accessed 8 April 2015.

29 McDonnell Max. Flappy Bird io Gameplay By The Numbers [online]. Flappy Bird io; 22 February 2014. URL: http://blog.flappybird.io/post/77513967646/flappy-bird-io-gameplay-by-the-numbers. Accessed 7 April 2015.

30 Flappy Bird [online]. Flappy Bird io; 2015. URL: http://flappybird.io/. Accessed 7 April 2015.

31 Robot Unicorn Attack [online]. Turner Broadcasting System, Inc; 2015. URL: http://games.adultswim.com/robot-unicorn-attack-twitchy-online-game.html. Accessed 7 April 2015.

32 Unity Documentation [online]. Unity Technologies; 2015. URL: http://docs.unity3d.com/Manual/index.html. Accessed 8.4.2015.

33 C# List<T> Class [online]. Microsoft; 2015. URL: https://msdn.microsoft.com/en-us/library/6sh2ey19%28v=vs.110%29.aspx. Accessed 13 March 2015.

34 Contra. Konami; 1987

35 The First Video Game? [online]. Stony Brook University, Battelle; 2015. URL: http://www.bnl.gov/about/history/firstvideo.php. Accessed 16 March 2015.

36 Alexander Leigh. Designing Better Controls for the Touchscreen Experience [online]. Gamasutra; 21 November 2013.
URL: http://www.gamasutra.com/view/news/205434/Designing_better_controls_for_the_touchscreen_experience.php. Accessed 3 February 2015.

37 Telfer Adam. Touch Control Design: Ways of Playing on Mobile [online]. 26 July 2014.
URL: http://mobilefreetoplay.com/2014/07/26/control-mechanics/. Accessed 3 February 2015.

38 Costikyan Greg. I Have No Words & I Must Design: Toward a Critical Vocabulary for Games. Tampere University Press; 2002.

39 Telfer Adam. Touch Control Design: Less (Control) is More [online]. 3 August 2014.
URL: http://mobilefreetoplay.com/2014/08/03/less-control-is-more/. Accessed 19 March 2015.

Evolution of Player.cs movement functionality

Excerpt from revision 1

```
float forceX = (Input.mousePosition.x - Screen.width/2)/Screen.width * 200;

float forceY = (Input.mousePosition.y - Screen.height/2)/Screen.height * 200;

Vector2 force = new Vector2(forceX, forceY);

rigidbody2D.AddForce(force);
```

Excerpt from revision 2

```
float v = joystick.position.y * speed;

float h = joystick.position.x * speed;

Vector2 force = new Vector2(h, v);

rigidbody2D.AddForce(force);
```

Excerpt from revision 4

```
if(Input.GetMouseButton(0)){
            float y = (Input.mousePosition.y - Screen.height/2)/Screen.height;
            if(y >= 0){
                        Up ();
            } else if (y < 0) {
                        Down ();
            }
}
```

Examples from Shark.cs

CheckTarget method

```
float CheckTarget (Transform target) {
            float playerDirection = transform.position.x - target.position.x;
            return playerDirection;
}
```

MoveTowards method

```
void MoveTowards (Transform target) {

            float playerDirection = CheckTarget (target);

            //Default move direction is to the right
            Vector3 moveDirection = new Vector3(1,0,0);
            if(playerDirection > 0){
                        moveDirection = new Vector3(-1,0,0);
            }
            transform.Translate(moveDirection * Time.deltaTime * movementSpeed
            * randomizer);
}
```

Jump checking logic

```
float playerDirection = CheckTarget (target);
if(playerDirection < 3 && playerDirection > -3){
            playerInRange = true;
            jump = true;
}
```

Examples from CollectibleSpawn.cs

TriggerSpawnEvent method

```
void TriggerSpawnEvent () {

            float spawnTime = Random.Range (spawnTimeMin, spawnTimeMax);

            StartCoroutine (Spawn(spawnTime));

}
```

Spawn method

```
IEnumerator Spawn (float time) {

            yield return new WaitForSeconds (time);

            int spawnRoll = Random.Range (0, 6);

            GameObject collectible = null; //Initiate collectible

            float spawnLocation = Random.Range (-15.0f, 15.0f);


            Vector3 spawnPosition = new Vec-
            tor3(transform.position.x+spawnLocation, trans-
            form.position.y,transform.position.z);

            //If 5, we spawn fat guy.

            if(spawnRoll == 5){

                        collectible = (GameObject)Instantiate(Resources.Load(
                        "Characters/Survivors/Collectible_fat" ), spawnPosi-
                        tion, Quaternion.identity);

            }else{ //If no special roll, spawn normal guy.

                        collectible = (GameObject)Instantiate(Resources.Load(
                        "Characters/Survivors/Collectible" ), spawnPosition,
                        Quaternion.identity);

            }

            collectible.SendMessage ("AssignSpawn", this.gameObject);

}
```

GUIButton.cs

```
public class GUIButton : MonoBehaviour {
	private GameObject player;
	public string command;
	public bool mouseDown = true;

	// Use this for initialization
	void Start () {
		player = GameObject.FindGameObjectWithTag ("Player");
	}
	void OnMouseDown () {
		if(mouseDown){
			player.SendMessage(command);
		}
	}
	void OnMouseDrag () {
		if(!mouseDown){
			player.SendMessage(command);
		}
	}
}
```

Scoring and Game Over behaviour

Rescued method in Collectible.cs

```
void Rescued (){
            if(collected && !dropped){
                        player.SendMessage("RemoveCollected",
                        this.gameObject);
            }
            gameControl.SendMessage("ScoreUp", value);
            GameObject scorePopup = Instanti-
            ate(Resources.Load("Prefabs/UI/ScorePopup"), transform.position,
            Quaternion.identity) as GameObject;
            scorePopup.GetComponent<TextMesh>().text = value + "$";
            GameObject.Destroy(gameObject);
}
```

SetScore method in Score.cs

```
public void SetScore (int score) {
            guiScore.text = "Score: " + score + "$";
}
```

ScoreUp method in GameControl.cs

```
void ScoreUp (int value) {
            points += value;
            guiScore.SendMessage("SetScore", points);
}
```

GameOver method in GameControl.cs

```
void GameOver () {

            //Activate gameOver on GUI side

            gameOver = true;

            GameObject.Destroy (playButtons);

            gameOverButtons = (GameOb-
            ject)Instantiate(Resources.Load("Prefabs/UI/gameOverButtons"), new
            Vector3(0,0,0), Quaternion.identity);

            GameObject.Find("winText").SendMessage("SetText","Game over!
            Score: " + points);

            //Get rid of spawners now that game is over, so no more spawning
            happens.

            GameObject[] spawners = GameObject.FindGameObjectsWithTag ("Spawn-
            er");

            foreach(GameObject spawn in spawners){

                        GameObject.Destroy(spawn);

            }

}
```

Parts of LevelGenerator.cs

Update method

```
void Update () {
            position = this.transform.position.x;
            if(position > index*interval){
                        index++;
                        int blockRoll = Random.Range (0, 5);
                        if(blockRoll == 1){
                                    SpawnRescue ();
                        }else{
                                    SpawnBlock ();
                        }
                        if(threatCountdown <= threatInterval){
                                    if(threatCountdown == threatInterval){
                                                Warning ();


                                    }
                                    threatCountdown++;
                        }else{
                                    SpawnThreat();
                                    RollThreat();
                                    threatCountdown = 0;
                        }
            }
}
```

SpawnBlock method

```
void SpawnBlock () {
            float spawnX = index*interval;
            float yRoll = Random.Range (-15, -5);
            Vector3 spawnLocation = new Vector3(spawnX, yRoll, 0);
            Instantiate(Resources.Load("Terrain/Block1"),  spawnLocation,  Qua-
ternion.identity);


}
```