An Animation & Particle Design Tool for a Location-Aware Mobile Game

by

Andrew James Fearon, BSc. Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

MSc. Computer Science

University of Dublin, Trinity College

December 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Andrew James Fearon

September 10, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Andrew James Fearon

September 10, 2010

I dedicate this to my family who have supported me without question.

Also to Sandra & my IET 2010 classmates who have kept me sane throughout the year.

An Animation & Particle Design Tool for a Location-Aware Mobile Game

Andrew James Fearon
University of Dublin, Trinity College, 2010

Supervisor: Mads Haahr

Current generation smartphones are quickly becoming a central hub for all kinds of activities besides communication. With a combination of powerful hardware, superb displays and open marketing channels, smartphones have seen a recent explosion of video game activity. This project was set in motion due to the restrictive nature of memory allowances on modern mobile devices and the increased demand for visual impact. The project required a visual effects tool for a location-aware mobile game allowing non-technical designers to create animations which would consume a minimal amount of memory on the device. The goals for the project were achieved through the combination of a desktop application which is used by designers to design animations and particle effects as well as an accompanying framework which executes the designs on the device. This ensured that designers were abstracted away from the technical level and could create engaging animations without the worry of large memory overheads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Mobile device are becoming increasingly used for a wider range of activities. No longer is a phone just a phone; they have evolved into much more than that and are now a central hub for a lot of daily tasks. With touch-screens, brightly lit, full coloured and detailed screens, GPS and powerful hardware underneath, phones really can do it all. With all of these capabilities games released on mobile devices are becoming more and more complex and the demand for quality is being raised for each release. With this advancement of quality mobile games require the same sort of tool support that current games require in order to be developed on time. This project was set in motion to develop tools that would aide the artistic designers for the Viking Ghost Hunt game to create memory-efficient visual effects for their game and in-game images.

1.2 Viking Ghost Hunt

Viking Ghost Hunt[27] (VGH), developed by the Dublin based Haunted Planet Studios [47] (herein referred to as 'the team'), is a location-aware mobile-game centered around the premise of moving through a city discovering historical clues that will lead you to solve a mystery. In the course of solving the mystery players will be educated on the history of the locations that they are visiting and will also be physically active as they walk between locations. Once players arrive at the location specified by the game

Figure 1.1: Players playing the Falkland Ghost Hunt at the recent trials in Scotland. Source: www.hauntedplanet.com

then are then challenged with a number of different tasks before they receive their clue. In some instances players must make use of an audio scanner in order to decode messages from characters while at other times they are required to use the phone's camera in order to scan the area, find the ghost and then 'capture' the apparition, this augmented reality approach within games can be very immersive for the player. The game is currently being developed for smartphones running the Android[19] platform.

1.3 Context and Problem

Since the game is played on a mobile handset the processing power available to the developers is limited and in particular these devices have very low memory allowances for applications. During development of VGH the team were finding that only a limited number of textures were fitting onto the device and that using sprite sheets as a method for animation was becoming overly expensive. Sprite sheets are a technique whereby a number of still images are composed into a single image and by drawing each single image, one after the other, you can simulate an animated image. Essentially

corresponding to the childhood method of drawing sequences in books and flipping the pages to create the illusion of smooth motion. As it became apparent that sprite sheets would not be a viable method for much longer it was decided that more of the built-in functionality of the handset should be utilised in order to bring life to the game's scenes.

1.4 Dissertation Objectives

This report is about how the device's built-in graphics pipeline was harnessed in order to give the characters and settings within the game more life while at the same time limiting the amount of memory using by the application.

This goal was accomplished using two separate methods. The first method provides a simple animation model to create basic movement and visual effects for a still image, which allows for movement of the texture within the game world as well as a few additional effects such as fading into and out of view. In addition to the animation model, a particle system on top of the animations can create a large degree of visual impact with minimal memory usage and an easy to understand structure. Particle systems have become very common additions to games in their short history as they can provide for a number of different visual effects with only a small set of rules that need to be tweaked to achieve drastically different results.

Both of these methods require a very small amount of memory in order to store the essential information required and so the device would be capable of storing more fundamental game related information as it was not committing all of its resources to carrying the overhead of animated textures. Further to this it was noted from the start that mission and art designers would be describing a lot of the animations and particle effects and so it was important that the lines between the creation of the animations or particle effects and those of the low-level programming required to execute them were blurred as much as possible. Overall the system would need to provide easy creation and use of animations and particle effects for non-technical users, while maintaining a low memory footprint.

1.5 Dissertation Roadmap

Here is a brief overview of the chapters that will follow in the dissertation and a quick description of what each chapter will contain.

Market and Research. This chapter will give an overview of the technologies used within the project; there will be a look into the smartphone market and Android's standing within it, as well as an in-depth overview of its architecture. The OpenGL ES technology will be investigated followed by a short look at previous animation and particle systems.

System Design. Covering the overall aims of the project this chapter will discuss the requirements of the project, and the high-level approaches that were taken in order to meet these requirements. The general animation method, its digital representation and finally the sketches of a user interface will be discussed.

Implementation. Starting with the proof of concept this chapter will break down the individual segments that comprised the framework and expand in more detail how they were developed.

Evaluation. Refactoring and profiling the framework in order to ensure smooth operation for the VGH team was important and this chapter will discuss some of the methods used in order to acheive this.

Conclusion and Future. In this final chapter the completed framework, its success in accomplishing the requirements set out previously and possible improvements will be examined. As well as this downfalls within the development cycle and alternative approaches that could have been considered will be mentioned.

Chapter 2

Market and Research

This chapter aims to provide an overview of the major technologies used within the project. A brief history of mobile games will be included in order to set the scene for the current state of mobile games. In order to familiarise ourselves with the target device the chapter will look into the smartphone market and Android's standing within it, as well as an in-depth overview of its architecture. This will be followed by a look into the technologies that are going to be used within the system implementation. The OpenGL and corresponding embedded systems APIs will be investigated followed by a short look at previous animation and particle system approaches.

2.1 Past, Present and Future of Mobile Games

Before discussing the most state-of-the-art aspects of mobile game development it may be useful to step back and have a look at what has pushed mobile gaming towards the success it is seeing today and the struggles that developers have had to overcome before. Mobile gaming has been progressing in leaps and bounds over the last 2 decades and is now showing promise as one of the most exciting fields in computer graphics. Mobile games are generally divided between two different categories of devices, dedicated hand-held systems and those that serve gaming as a secondary interest. Handheld gaming system started gaining popularity with the Nintendo Gameboy system[38] when it was released in the late 1980s (Japan and America) to early 1990s (Europe) which brought popular game characters into the hands of more people than ever. These system were

not overly powerful with general specifications reaching roughly 4-5MHz processors with less than 10kB of internal RAM. However with games such as Tetris[35] and Pokemon[11] gaining massive popularity and driving sales of handheld devices into the millions[37] the mobile industry had set its feet solidly underneath the gaming umbrella and would be there to stay.

With the widespread adoption of mobile phones, one of their first ulterior uses beyond telecommunications was entertainment in the form of video games. These games were quite simple as the screens were still very simple pixel matrix displays. Games like Snake[39] for the Nokia 6110 gained widespread popularity among teenagers and young adults and set the groundwork for a new generation of mobile phone games and players. Snake was designed by Taneli Armanto, a design engineer for the user interface software at Nokia and at present it is estimated that there are 350 million phones in circulation with Snake already embedded on them.

Handheld video game consoles have now reached the stage where they are roughly one or two generations of house-hold console iterations behind in terms of processing power and are selling in equal amounts. Take for example the port of Super Mario 64 onto the Nintendo DS; Super Mario 64 was released in 1996 to coincide with the release of the Nintendo 64 and only 8 years later, a small number of years after the release of the GameCube and before the world would see the release of the Nintendo Wii, Super Mario 64 DS was available. The PlayStation Portable has seen an even more marked improvement in terms of the graphics that can be rendered to the display and the overall number crunching power available to developers. The PSP is estimated to be able to perform about on par with the PlayStation 2, a console still in widespread use and the highest selling console of all time[50].

While mobile-games for phones had seen improvement over the years between their initial break onto the scene and the decade after, the format was never a major success for developers. The reasons for this were numerous, device hardware was different between each device and so an application would have to be re-written in order to work across all of the different hardware setups, games were generally developed in-house by the team that was developing the operating system (such as Taneli Armanto's Snake) that would be used on the phone and at this stage there were no unified operating systems.

With the advent of Java-capable phones developers began to explore the possibilities

Highest ranked paid applications - Apple App Store for iPhone (US, June '10)

Rank	App	Publisher	Category	Price
1	Angry Birds	Clickgamer.com	Games	$0.99
2	Doodle Jump - BE WARNED: Insanely Addictive!	Lima Sky	Games	$0.99
3	Fruit Ninja	Halfbrick Studios	Games	$0.99
4	BATTLESHIP	Electronic Arts	Games	$2.99
5	FIFA World Cup™	Electronic Arts	Games	$4.99
6	FatBooth	PiVi & Co	Entertainment	$0.99
7	TETRIS®	Electronic Arts	Games	$4.99
8	Pocket God	Bolt Creative	Entertain-ment, Games	$0.99
9	Skee-Ball	Freeverse, Inc.	Games	$0.99
10	Guitar Hero	Activision Publish-ing, Inc.	Games	$2.99

Figure 2.1: App store report from July 2010. Source: www.distimo.com

of charging people to download their games. This solved many problems as the Java implementation was generally capable of running on any device that supported the Java framework, however, the marketplace for games was still very fractured. People that wanted to play these games often had to browse highly unreliable or suspect websites, sign-up for large monthly fee schemes, etc... on top of this the quality of the games was not monitored which lead to a lot of reluctance for people purchasing the games.

Modern phones have broken down almost all of the barriers that have stopped mobile video games from being a profitable investment before. With modern operating systems such as iOS or Android the users are automatically connected and linked into the operating system's marketplace, App Store[3] for the Apple phones and the Android Market[21] for Android phones. These market places give clearly defined central locations for developers to promote, sell and market their products and at the same time ensures that users who are buying the game are fully aware of what they are buying and can more easily trust the source.

With these new nodes for selling games and with the incredibly powerful devices which are now reaching users, games have exploded in popularity on mobile devices and have finally begun to reward developers with significant monetary gains[48]. Even indie developers can now enter the market as there are very few barriers blocking the

way. As it stands today, of the top 10 paid apps sold on the apple app store, over 9 out of ten of those are games[10][4] as seen in Figure 2.1. It is clear that now is a great time for individuals, small teams and even large corporate companies to be pushing their fresh ideas or existing franchises into the hands of more and more people. While a recent report[9] in the US showed that there had been a 13% decline in mobile gaming it also showed an impressive 60% increase in mobile gaming on smartphones, likely due to the previously mentioned complications of playing games on non-smartphones. 47% of people surveyed said that they played a game on their smartphone at least once a month, and the majority of non-smartphone users said they only played the games which had come preloaded onto their phone. Recently Steve Jobs of Apple has come forward to recognise the unexpected significance of games on the App Store[15] and Apple's potential to compete with Nintendo and Sony.

The industry reports as a whole are very promising for the mobile market yet some analysts[8] are comparing the present mobile app market to that which was seen in the days of the dot com boom, a market with drastically reduced entry requirements offering a world of promise and potential sales. With this, many companies and individuals are spending a lot of money committing projects for mobile devices that have perhaps not been fully thought out. Hopefully this will not be the case but there is certainly potential for another realisation that not everybody is going to make millions in the mobile market.

The future remains bright for mobile devices as demonstrations (as seen in Figure 2.2) from two of the gaming industries largest companies, id Software[12][2] and Epic Games[31][32], have pushed the hardware available beyond the limits that had previously been envisaged. Using their extremely adaptable desktop application engines, both id Software and Epic Games have shown that the semantic gap between mobile, living room and desktop gaming is closing quickly. Apple's recently launched iPad is another major game development platform and many development studios have taken an interest in making games for it. With the current trends it seems that mobile gaming is at the forefront of gaming advances and that this is something which will be around for the foreseeable future.

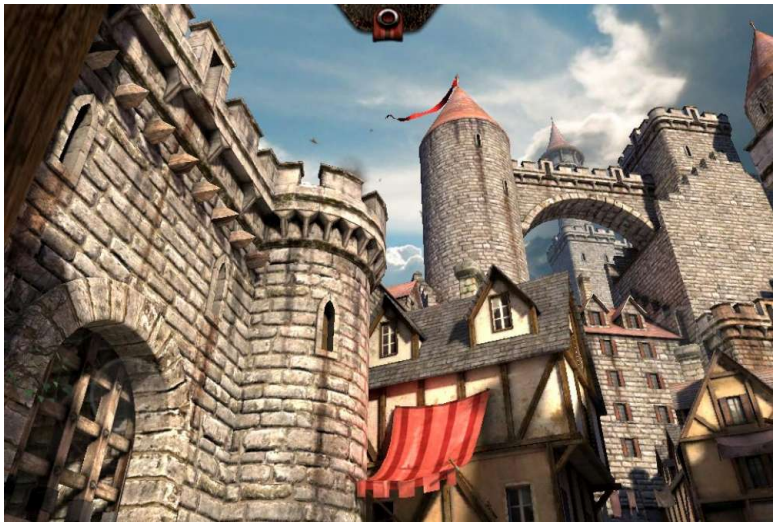

Figure 2.2: Recent demo of Epic Games' Unreal Engine on the iPhone. Source: www.kotaku.com

2.2 Android Overview

Released on February 9th of 2009, Android is a mobile operating system originally developed by the company Android Inc who were then subsequently bought by Google Inc. The Android operating system is one of the modern smart-phone operating systems which include; iOS, the operating system which apple has developed for the iPhone, Windows Mobile, Symbian (NOKIA), and Research in Motion (RIM) the operating system used on BlackBerry devices.

A report from Gartner on August 12th of 2010[16] indicated that the spread of operating systems within the mobile phone market is still very much in flux. While this competition within the market has been good for consumers as they are seeing decreased costs in entering the latest generation of mobile technology it is difficult for developers as they must target a specific operating system or be prepared to port their application onto multiple different operating systems, each of which requires its own specific considerations.

As seen in Figure 2.3 the market is quite divided but the four main operating systems available, with a share of over 90% of the market are, in order, Symbian, RIM, Android and iOS. Symbian takes such a lead as it has been around for quite a long time and has been embedded into a huge number of handsets within the market. RIM

Company	2Q10 Units	2Q10 Market Share (%)	2Q09 Units	2Q09 Market Share (%)
Symbian	25,386.8	41.2	20,880.8	51.0
Research In Motion	11,228.8	18.2	7,782.2	19.0
Android	10,606.1	17.2	755.9	1.8
iOS	8,743.0	14.2	5,325.0	13.0
Microsoft Windows Mobile	3,096.4	5.0	3,829.7	9.3
Linux	1,503.1	2.4	1,901.1	4.6
Other OSs	1,084.8	1.8	497.1	1.2
Total	**61,649.1**	**100.0**	**40,971.8**	**100.0**

Figure 2.3: Worldwide smartphone sales to end users by operating system in 2nd quarter 2010 (Thousands of Units). Source: Gartner (August 2010)

have seen quite a lot of success with the BlackBerry handset but its popularity may be slowly decreasing as the figure shows with a small loss of 0.8%.

Of most interest within the chart are the figures for both the Android and iOS platforms; both platforms have seen large growth within a short timespan. iOS being released in 2007 has taken a large portion of the market and Android has moved up to almost 20% of the market within a year. A furture report from the NPD Group[26] has shown that Android is now taking the lead in sales in the US. This would point towards Android as being the platform of choice for developing applications as it has such a fast expanding userbase but that is not the only factor to take into account when choosing a platform.

2.2.1 Architecture

A video overview of Android's architecture was released by Google in 2007 detailing a lot of underlying intricacies of the design[17] (Figure 2.4). Android was designed as an open software platform for mobile development; the aim was to provide a full operating solution from the basic process and resource management right up through middleware and onto applications. The underlying driver and hardware link is built from a Linux 26 kernel foundation.

Linux was chosen as it provided a proven and stable driver model which has all of the most commonly required interfaces such as input and output (I/O) , memory

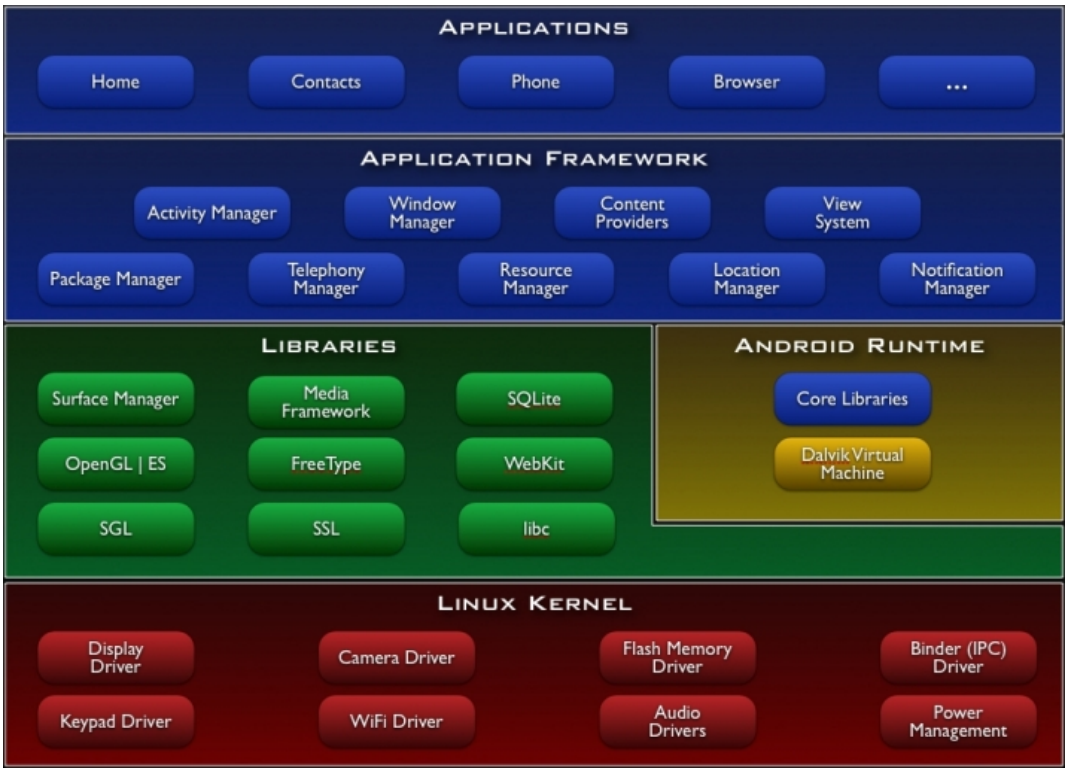

Figure 2.4: Overview of the Android architecture showing 4 different sections. Red; Linux Kernel. Green; Libraries. Yellow; Android Runtime. Blue; Applications and Framework. Source: www.android.com

management etc..., drivers and implementations which have been tried and tested over a number of years. This strong background of stable driver implementations also means that there are a large number of drivers already available for various hardware setups. The back catalogue of drivers and their extensive use helped in making the initial setup up of the embedded device a lot easier and ensured a more robust implementation.

After the lower level of drivers there are a series of runtime libraries used for the fundamental operation of the phone, the code for these classes mostly consists of c or c++. Included within these libraries are objects for media playback, compression and encoding, with all the major formats supported. Along with this are font libraries, an SQLite implementation and Webkit which is used as the Android browser engine, the same engine used by Apple's iPhone.

The library of most importance to this project however is the Surface Manager. The Surface Manager is responsible for correlating all of the currently active windows and the views within them and ensuring that the correct composition of pixels is drawn to the screen. It will be within these views and through overwritten views that the graphics for this project will be displayed to the user and so we will be interfacing directly with the Surface Manager.

Dalvik Virtual Machine

One of the crucial aspects of the Android architecture is the runtime environment and specifically the Dalvik Virtual Machine (VM). Dan Bornstein has been the mastermind behind the Dalvik VM, on which he has been working almost exclusively since he moved to Google in 2005[7]. A VM is the layer which which sits between the code that a developer has written and compiled and that which will be run on the underlying hardware. Java's use of VMs has been one of the features that make it appeal to developers. The universal nature of VMs has been captioned by Sun Microsystem as 'Written Once, Run Anywhere'. This is a very useful tool when device manufacturers need to ensure that the code written by developers can be run on a number of various different hardware setups. However the draw back with the use of VMs are that the implementation for different hardware setups may cause the program to behave in slightly different fashions depending on the model that it is executed on. This then leaves the developer with the task of testing the program on multiple setups to ensure

that the program is behaving as expected across all targeted platforms.

The Dalvik VM was meet with a number of challenges as it was targeted towards a mobile device. These challenges included;

- Run on a slow CPU.
- Run with relatively low RAM.
- Run on an OS without swap space.
- Run on a device powered by battery.

Each of the items presents different degrees of complexity to the VM designer and all ask a lot of the final VM. With slow processing speeds it is important that the code running on the VM executes as fast as possible. The combination of low RAM and no swap space (the technique of offloading sections of RAM onto slower secondary storage devices) demands that the code is consuming as little space as possible.

Finally, the fact that this is all to be run on a device powered by a battery is a huge point of interest both to the VM itself and any developers looking to write applications for the device. It has been an issue for many years now with mobile devices as the chemical science behind battery life has struggled to maintain equality with the Moore's Law growth of the semi-conductor world. This demands that while the VM is executing instructions as fast as it can manage and at the same time ensuring that it does not overuse the sparse memory supply of the device, it must also do so in as efficient a manner as possible in order to prevent draining the devices power in an instant; no mean feat.

Typically, the Java VM works by translating the code written by a developer into bytecode, this code is then run through the VM and executed on the underlying hardware. Dalvik made most of its changes by taking the bytecode generated for the regular Java VM which has been converted into .class files and attempts to compact these files in a number of ways, the result of which is stored as a .dex file.

The focus of the approach is to try and reduce redundancy in the .class files and this is typically done in two ways. Firstly, multiple class files are compared to one another and any shared strings between the two classes are placed into a single .dex file. These strings can be literal strings, method calls or similar. Secondly, typed objects are

	Common System Libraries	Web Browser	Alarm Clock
Uncompressed Jar	100%	100%	100%
Compressed Jar	50%	49%	52%
Uncompressed Dex	48%	44%	44%

Table 2.1: Dalvik file sizes verses generic compressed and uncompressed files.

pooled into common areas designated by their type. Thus, objects are implicitly typed by the area in which they are located saving unnecessary typing.

As is shown in Table 2.1 the compression rates are extremely favourable over uncompressed jar files and in places significantly favourable over compressed jar, while in all cases it provides at least a 2% decrement in size. These compression rates have great benefits in terms of the memory usage on the devices small memory capacity.

The CPU used in a typical Android device is roughly similar to a 10 year old desktop machine (hardware setup will be examined in Section 2.2.2) and so a few tricks are employed within the dex implementation in order to speed up the CPU efficiency. Along with slower processing speeds of the CPU, there was also still the memory issue whereby the CPU had a very small cache in which to store instructions and data.

One of the significant benefits of the platform is that the libraries at the essential functionality level are all written in native c/c++ code and so the vast majority of end calls are being performed with this native code implementation. This leaves developers with the chance to write the logic of their applications in a higher level language such as Java and they can be assured that the low-level functional calls will be executed quickly.

The Java Native Interface(JNI) has been made available to android developers should they wish to squeeze every ounce of performance from the device that they can but the Dalvik designer Dan Bornstein insists that most speed ups will be gained from clever programming in the higher level languages by trying to ensure that your program is sleeping for the majority of its lifetime and that it loops intelligently when it must.

Changes made at a deeper level within the interpreter, right down at the assembly code level were only slight changes but made for significant improvements. Firstly, when applications are installed onto the Android device the dex file is verified during the process and this saves time during the application's execution as it saves any

security or typing checks from having to be performed. In addition to this the dex file is optimised and augmented according to the device that it is installed on.

Static linking replaces the string literal names of methods or fields with an integer offset for a table. An integer offset is much faster to calculate than searching for the matching string literal for a method name which shaves time off the execution of instructions. This final dex representation in terms of low-level assembly instructions has the following properties over the standard compiler. The final item of 35% more bytes in the stream is overcome due to the fact that the interpreter can consume two bytes at a time.

- 30% fewer instructions.
- 35% fewer code units.
- 35% more bytes in stream.

The first number of Android iterations explicitly excluded the inclusion of a Just-In-Time (JIT) compiler as there was not enough memory to store the bloated assembly code base when compiling on the fly. It was an unfortunate loss to have to sacrifice but in the most recently released Android version Google have included a JIT compiler which has seen very significant increases in the processing speed of their CPU, as seen in Figure 2.5.

Supporting the Dalvik VM are the core libraries which offer the typical classes and objects that a Java developer would expect such as collection classes for storing lists of objects, I/O objects for reading or writing files and other utility classes for the vast majority of functionality supported by a regular Java API. The libraries are not a complete representation of the Java API in order to try and save more space on the device. The main absence from the API are Swing and JWT implementations as Android has its own set of libraries for user interfacing.

Unfortunately it has not been plain sailing for Google and the Dalvik VM. In August 2010 Oracle, who had recently purchased Sun Microsystems and so now own the Java property, sued Google over their alterations to the Java VM[40].

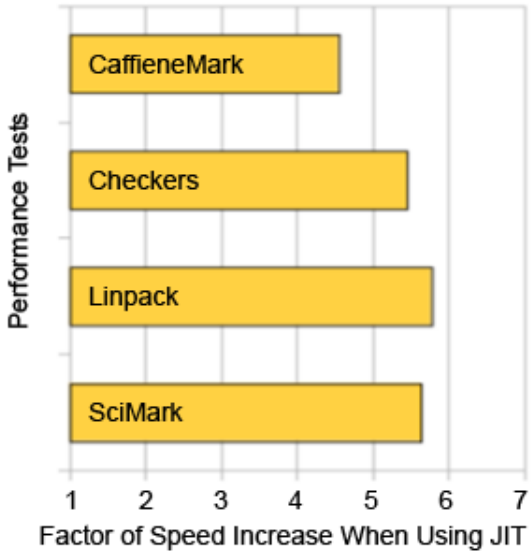

Figure 2.5: Speed increases using the Android 2.2 JIT. Source: www.android.com

HTC Model	Release Date	Screen Dimensions	CPU	ROM	RAM
Dream	October 2008	320x480	528MHz	256MB	192MB
Hero	July 2009	320x480	528MHz	512MB	288MB
Desire	March 2010	480x800	1GHz	512MB	576MB

Table 2.2: Progress in HTC hardware over Android's lifetime. Information from www.htc.com

2.2.2 Hardware

One of the separating factors between development for an Android device and that of its nearest rival the iPhone is that iOS has been developed for a strict set of devices and hardware layouts. These layout have been designed by Apple themselves so both OS and Hardware are very tightly integrated and controlled. Throughout its short lifespan, the Android OS has been run on a number of different hardware layouts while the iPhone has seen 4 strict iterations.

The first device to market that would run Android was the HTC Dream from T-Mobile and since then there has been a number of releases from different manufacturers including Samsung, LG and Motorola. A quick comparison of the releases from just HTC alone (Table 2.2) will begin to show the changes that have taken place over the short period of Android's life.

While the latest generation are beginning to increase the size of the display as well as including a more powerful processor, it seems clear from the beginning of the hardware iterations that memory was the largest issue for the first generation of phones. While the hardware designers are obviously trying to alleviate any bottlenecks that they can, memory limits still place huge restrictions on large or complex programs. This is one of the reasons why the team has come forward to request this project as a reduction in the memory footprint of an application can have a huge impact on the user's experience and also ensures a level of backwards compatibility with older models. This memory trend is also seen in the iPhone iterations which have moved from 128MB in the initial model to 512MB in the most recent.

2.2.3 Version History

So far Android has seen 4 major version updates since its inception and we will cover them briefly here[18]. Released in April 2009 Android 1.5 brought with it a large amount of alterations but most significantly for this project, Android 1.5 was the first version with the GLSurfaceView implementation and as such would allow a very easy interface framework for developing OpenGL applications. Android 1.5 is the minimum supported version at present and is gradually losing market presence as seen in Figure 2.7.

Android 1.6 also contained some interesting additions for the project as it included animation classes for various interpolations. These interpolator classes would be used later on in the design and implementation phase in order to add more variety to animations without sacrificing a lot of memory. While not immediately significant to this project as it was undetermined what version of Android would be targeted by the final game, version 2.1 added support for OpenGL-ES vertex and fragment shaders. These shaders open up an entirely new dimension of low-memory alteration potential and will be discussed at a later stage.

Most recently Android 2.2 has launched in May 2010 and, as mentioned in the Dalvik section (Section 2.2.1), the change most applicable to this project was the new JIT compiler for the Dalvik VM. Google took the game Replica Island as a test case for the new interpreter and was able to show just how much of a performance increase can be gained through the use of the new interpreter.

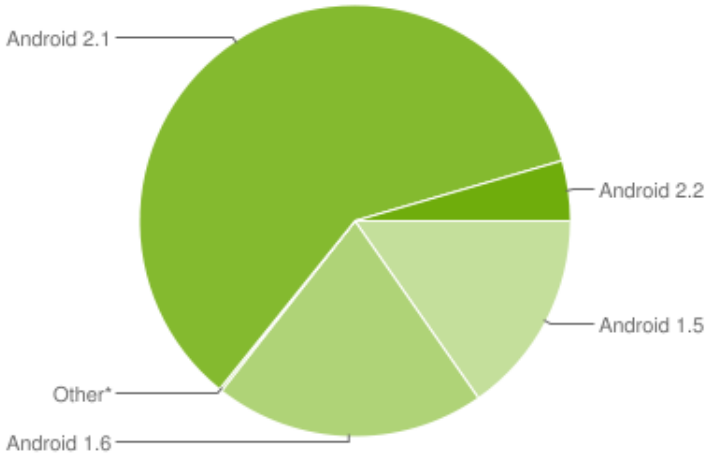

Figure 2.6: Data collected during two weeks ending 2nd August 2010. Source: developer.android.com

The small yet important changes between the different versions of Android have had a significant impact on the performance and possibilities within the games that are run on them. It is of course always desirable that you are running on the latest platform that can offer the fastest processing as well as the greatest variety of options, however, not every device in the market is capable of running the latest version and not all the handsets that are capable of the latest version are in fact running it. To illustrate the current market Google has released a couple of charts (Figure 2.6 & Figure 2.7) to aid developers in their choice of platform. With a share of almost 65% of the market and that share quickly rising, versions 2.1+ look very promising for modern and graphically intense games to target, while simpler games may well be able to target a larger share of the market.

2.3 OpenGL ES

There are two major players in the world of hardware accelerated 3D graphics APIs, those two players are DirectX[36] and OpenGL. DirectX has been developed and supported by Microsoft since its first release in 1995. DirectX is used to create almost all computer games written for Windows and has been without competition in that area for a long time now. OpenGL on the other hand has been used to write the vast

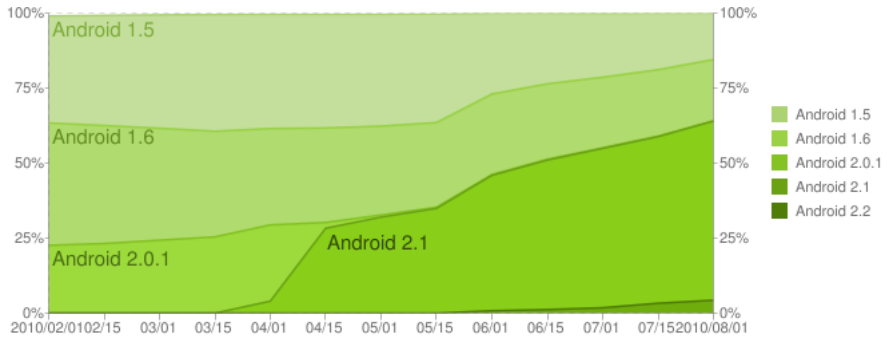

Figure 2.7: Historical spread of platform versions. Source: developer.android.com

majority of non-gaming graphical applications for quite some time now. OpenGL is the preferred API for the majority of applications as it is an open platform and so has seen implementations that will run on all modern operating systems such as Linux, Mac and Windows.

OpenGL has been developed by the Khronos Group[25]. The Khronos Group, as described by their website are 'An industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices'. As a natural extension to the OpenGL API the Khronos Group have developed the OpenGL Embedded System (OpenGL ES) standard. As the name implies this API is aimed at embedded devices and so it aims to help minimise the problems of these devices as have been mentioned before, such as their lower memory storage, slower processing speeds, slower bandwidth and finally the power consumption issue.

One immediate difference between the OpenGL API and its ES counterpart is the lack of redundancy within the OpenGL ES API[1]. This lack of redundancy aims to remove options from the developer that would allow them to do the same thing in different ways. One of the most commonly highlighted examples of this is the draw methods available between OpenGL and OpenGL ES. In OpenGL you can draw vertices using immediate mode, display lists or vertex arrays, while in OpenGL ES the former two have been removed and so the developer must use vertex arrays. The immediate mode is a very nice feature of OpenGL as it allows a very inexperienced programmer to draw vertices to the screen quite quickly but this ease of use is costly in terms of memory efficiency and the redundant methods also serve to bloat the APIs code base.

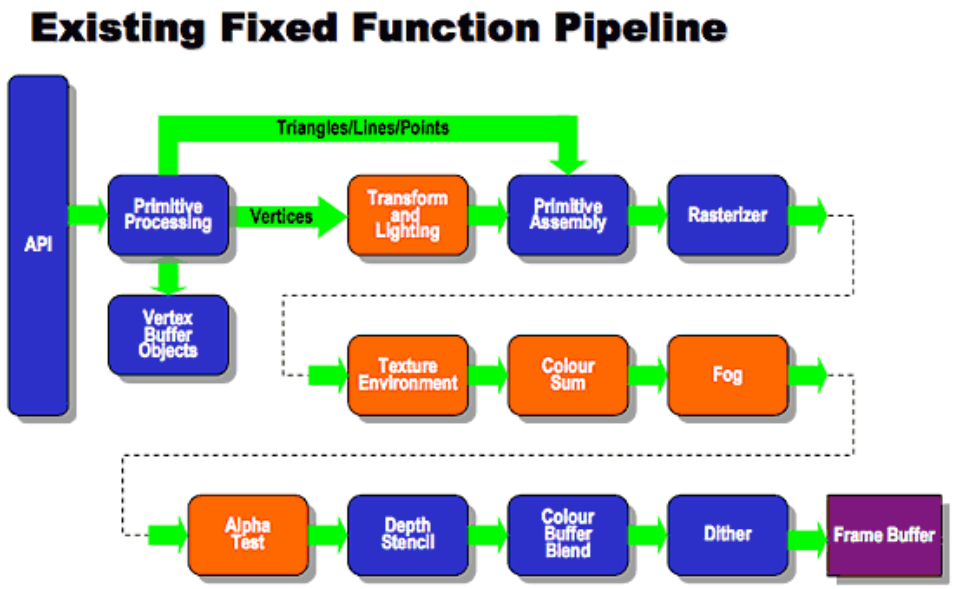

Figure 2.8: OpenGL ES 1.x with fixed function pipeline. Source: khronos.org

While developing the OpenGL ES API, it had been the goal of Khronos that they could provide an API that would be compatible with the standard OpenGL API and while this has been mostly achieved the reduced redundancy has unfortunately had an impact in just how compatible the two are. A very significant amount of the two APIs are compatible however and this allows developers that have already written code using OpenGL to transfer it to embedded systems with minimal hassle and if the code is newly developed it is quite easy to determine if the code will run on both OpenGL and ES APIs as you develop.

OpenGL 1.x Extensions

Due to OpenGL's collaborative ideology it is open to extensions. Extensions are implementations which are added into the OpenGL API by external hardware manufacturers in order to support new and specific methods within their pipeline. This ability to extend the underlying OpenGL API can be extremely useful as it allows for manufacturers to constantly push the limits of their hardware and not have to wait for a software interface first. The downside however is that while an extension may be supported by one device it might not be supported by another.

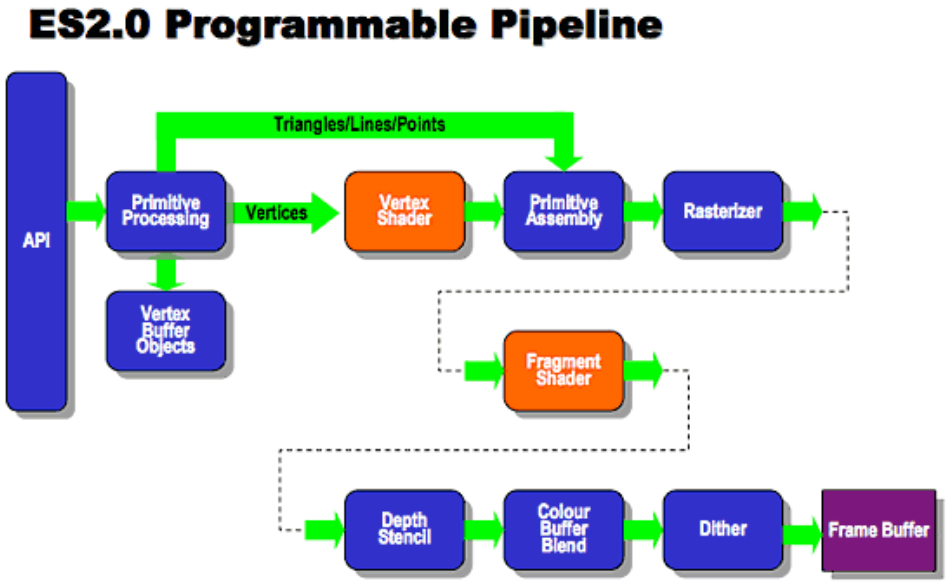

Figure 2.9: OpenGL ES 2.0 with programmable pipeline. Source: khronos.org

This method of extensions is counter to DirectX's approach as DirectX is solely maintained by Microsoft and so the functionality is set with each release. Extensions within OpenGL ES 1.x present a big issue for developers who are trying to ensure that their software will work on as many devices as it can as they cannot be sure that all devices in the market will be able to use any extensions that they have enabled. The list of extensions to ES 1.x is quite extensive[24] and while it allows for newer techniques to be available in a shorter period of time it has resulted in compatibility issues that the Khronos Group was trying to avoid.

OpenGL ES 2.0 Programmable Pipeline

The most significant change that came around with the OpenGL ES 2.0 pipeline implementation was the introduction of a programmable pipeline. The effect of this change on the pipeline can be seen between Figures 2.3 and 2.3. The programmable pipeline allows developers to write their own code for the fragment section of a graphics pipeline. It used to be the case with desktop applications, and has been the case with mobile devices until recently, that in a graphics pipeline the developer will have no say in

terms of how the fragments of a graphics pipeline are manipulated or finalised.

Essentially, a fragment is a pixel that has yet to be decided whether or not it will be shown. In a fixed pipeline a fragment's fate is typically decided by simple things such as a depth buffer or alpha fading, thus if the depth buffer is turned on then the fragment which is closest to the camera will be the one that is chosen as the pixel to be drawn, while without a depth check then it is typically a random decision and this can result in behaviour called z-fighting where pixels alternate every frame with regards to which pixel is to be drawn on-screen.

The ability to manipulate these fragments before they are discarded by the graphics card has seen major improvements in terms of techniques and visual effects for video games and computer generated graphics in general. Until recently programmable pipelines tended to offer two stages that a programmer could manipulate, the vertex layer and the pixel or fragment layer. At the top of the line DirectX 11 cards there is now an additional geometry shader which allows programmers to alter the geometry of an object before it is passed to the vertex shader but this layer is a long way from being relevant at the mobile level so we will not discuss it beyond mentioning it's existence.

At the vertex layer programmers can write code to manipulate the location of objects, compute direction vectors, etc... and these values will then be interpolated down into the pixel shaders. At the pixel or fragment shader level the programmer can opt to stick with the generic interpolation from the vector shaders which will result in very fast calculations but reduce the complexity of techniques, or the developer can continue techniques at the fragment level allowing for much greater precision and detail within the shader.

Programmers use a shader language specific to OpenGL ES in order to write the shaders that will control the vertex and fragment sections of the pipeline. These languages are quite low level but support the functionality required within a graphics pipeline. Basic types are included such as booleans, integers, floats and matrices and then on top of this a collection of mathematical methods are given to the developer as these will prove invaluable in calculating distance vectors, angles between vectors or ensuring that a float variable is clamped to within certain values.

```
private final String mVertexShader =
    "uniform mat4 uMVPMatrix;\n" +
    "attribute vec4 aPosition;\n" +
```

```
    "attribute vec2 aTextureCoord;\n" +
    "varying vec2 vTextureCoord;\n" +
    "void main() {\n" +
    "  gl_Position = uMVPMatrix * aPosition;\n" +
    "  vTextureCoord = aTextureCoord;\n" +
    "}\n";

private final String mFragmentShader =
    "precision mediump float;\n" +
    "varying vec2 vTextureCoord;\n" +
    "uniform sampler2D sTexture;\n" +
    "void main() {\n" +
    "  gl_FragColor = texture2D(sTexture, vTextureCoord);\n" +
    "}\n";
```

The code above is an example of a vertex shaders and a pixel shader supplied as a sample by Google for Android. The code for the shaders is being wrapped in a simple *String* for this example but it is a good example for illustrating the technical nature of shaders. A person creating a shader must be acutely aware of the information that will be passed to the shaders and they must be aware of how it will be passed. If a developer is aware of this then there are a wide variety of possibilities that are opened up to them, however, the developer must have the knowledge and understanding to do this.

This is the main reason that shaders were not focused on as part of this project as the requirements for Viking Ghost Hunt were for a system that could be used by designers, and unfortunately shaders are of quite a high technical level and as such did not fit into the context of this particular project. As will be discussed in future considerations (Section 6.3) shaders are certainly something that should be looked at for the Ghost Hunt games as they will offer a substantial graphical and performance boost to the game, but the problem that will arise is that they will need technical people to work with the designers in order to write these shaders.

2.4 Animation and Particle Effects

Animation and Particle Effects are two of the most fundamental approaches used in order to create a more vibrant world for the players playing a game[49]. Animation adds movement to simple static objects and brings life to previously still and lifeless scenes. Similar to this, particle effects can add a lot of spark to scenes for very little cost. Both techniques can also be used to attract the players attention to a certain object or area within the game.

2.4.1 Animation

Animations are typically created in one of two ways, either through a user interface such as Blender[5] or with some form of descriptive language. Descriptive languages can range from languages fashioned purely for the description of animations[46] or more abstracted and established languages such as ActionScript for developing Flash animations. One thing that all of these styles have in common is that the systems revolve around key-framing, key-framing serves to cut down on the information required to be stored as only significant events in the animation are noted. How this works will be looked at now.

Key-frame Animation

Key-Frame animation has been a staple in all types of animation for many years now, as far back as the pen-and-paper beginnings of animation[52]. Key-frame animation essentially involves defining key points within the actions of an animations and then the blanks can be filled in between the two key moments. This is done in pencil drawing for example when a character performs an important action, perhaps the character is bending over to pick something up. The lead artist will draw perhaps the initial shot of the character standing straight, a half-step from standing to picking up, then shot of the character physically picking up the item and then the return to standing shot.

These 5 small shots should not take too long for the artist to draw and give a good sense of what the animation should look like. Newer artists will then be tasked with the job of filling in the gaps between the key-frames and this is not too difficult as they have a clear beginning and end point that they need to achieve and they mostly need

to worry about consistency, smooth transitions and volume preservation. Once this is complete there is now a smooth animation from start to finish of the character picking up the object, the essential feel of the animation has been thought out and drawn by the lead artist while the interim frames have been drawn by others.

This same approach is what will be used for the animation designer of the project. The designer will describe a few essential moments or actions that they wish to convey to the players and the framework will fill in the gaps between the key-frames. This ensures that, similar to traditional line drawings, the artist can describe the feel and animation that they want to achieve without needing to painstakingly spend all of their time describing the minute details of each individual frame.

2.4.2 Particle Effects

The Particle Systems paper by William T. Reeves[44] was the first mainstream uses of particle systems, it showed the visual power that could be achieved with a set of very simple rules which could be altered to suit requirements. Most particle systems are based around the following simple variables for each particle.

- Position.
- Velocity.
- Size.
- Color.
- Opacity.
- Lifetime.

When a specific particle within a particle effect is created it is given a random value for each of these different variables and then that individual particle is simply updated and allowed to go about its 'life'. Once the particle's lifetime has expired it is reincarnated with a new set of values and the cycle repeats ad infinitum. The strength of the particle systems is the number of these simple objects that can be drawn at once and once there are enough being drawn then the futile life of a single particle is

no longer obvious to the viewer. While at the same time, having such a large number of particles on screen, each having slightly varying yet recognisably similar behaviour, allows for a very natural feel.

The initialisation of the particle can have more constraints placed on it in order for the particle to exhibit certain behaviour; for example there may be limits placed on the magnitude of the initial velocity (thus limiting how fast the particle can move initially), or there might be constraints placed on where particles can be placed at the beginning of their lifespan i.e. they may only be placed around the perimeter of a rectangle or they may only lie on the face of a sphere. These simple constraints allow for a very significant range of variety while still maintaining very believable and natural effects; of course the particle systems are not always used for realistic effects and may just be used to signal the players attention or provide a visual confirmation of an action they have performed.

Viking Ghost Hunt had similar requirements for a particle system; it would add a nice visual touch to the gameplay but could also serve as a way for designers to draw the players attention to ghosts that might be close or simply as another visual cue that they had pressed a button on the interface. While it is understandable that Android does not offer a built in particle systems in order to save any memory possible, this then meant that a particle framework would need to be developed for the Viking Ghost Hunt team for use in their game.

Chapter 3

System Design

Starting with the requirements set out by Haunted Planet Studios this chapter will describe the high-level approaches which were taken in order to meet them. This will include the process behind determining which data representation would serve best as an intermediate layer between the designers and the device software. As well as this there will be discussion on the conceptual beginnings of the user interface for designers, including a method for simplifying the spatial representations for designers creating augmented reality animations.

3.1 Requirements

The main requirements for the project have been compiled into the following list.

- Improve ease of animation within the game.
- Make spacial animation easier.
- Ensure that the solution caters to designers.
- Offer ways to increase visual appeal.
- Minimise memory usage.

After hearing the initial requirements of the team it seemed clear that the ideal solution to the problem would involve a straight forward and easy to use Graphical

User Interface (GUI) for the designers. A simple GUI would allow the designers to design and view the animation before transferring their work on to the phone in order to determine if it was what they wanted. This was quite a lofty ambition straight off the bat and so the design and requirements were broken down into more simplistic steps which would hopefully lead towards the final goal that had been envisaged by piecing them together.

The first and most basic need for the system was the way to represent the animations and what exactly it was that would need to be presented and exposed to the designers. It was agreed by everyone that the essential requirements of the animation system should provide ways to describe the position, scale, alpha/opacity and rotation of a game object. With these fundamental needs in place we could begin to look at how to represent them in the game.

3.2 XML and Animation Descriptions

When the team had initially proposed a collaboration the goal was to build a mission scripting system that would allow mission designers to craft exciting scenarios while again being abstracted away from the technical level. However the needs of the team changed over time and so ultimately the mission scripting was scrapped in favour of the animation tool. During the research period for the mission scripting however there had been a lot of work done looking into Lua[34][33] and it was suggested that this might be a suitable candidate for describing the animations.

This seemed inappropriate however as an essential requirement of a Lua implementation would automatically oppose the requirement for a low-memory solution. The reason for this is that since Lua is an interpreted language a further interpreter would need to be packaged with the game application in order to execute the Lua commands. This addition would then drain more resources from the developers small memory allowance and perhaps bring too much power to solve a problem that was easier to solve at its core than the previous mission scripting idea.

Extensible Markup Language (XML)[51], is a simple data description and markup language which was first exposed to the development community in 1996 and since then has seen widespread use over the last number of years in a huge number of applications. XML's strength lies in its simplicity; the language demands clear entry and exit points

for data information and so it is easily human and computer readable. With the small number of variables that were required to be represented within the animation description and its clear data structure a text based representation seemed like the next logical step, and in fact, this is as it turned out how Android were doing their animations for the 2D views. The following XML from one of the ApiDemos is used to stretch, then simultaneously spin and rotate a View object.

```
<set android:shareInterpolator="false">
   <scale
      android:interpolator="@android:anim...
         .../accelerate_decelerate_interpolator"
      android:fromXScale="1.0"
      android:toXScale="1.4"
      android:fromYScale="1.0"
      android:toYScale="0.6"
      android:pivotX="50%"
      android:pivotY="50%"
      android:fillAfter="false"
      android:duration="700" />
   <set android:interpolator="@android:anim...
      .../decelerate_interpolator">
      <scale
         android:fromXScale="1.4"
         android:toXScale="0.0"
         android:fromYScale="0.6"
         android:toYScale="0.0"
         android:pivotX="50%"
         android:pivotY="50%"
         android:startOffset="700"
         android:duration="400"
         android:fillBefore="false" />
      <rotate
         android:fromDegrees="0"
         android:toDegrees="-45"
```

```
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:startOffset="700"
            android:duration="400" />
    </set>
</set>
```

Using XML as the container for describing the animations allows for very clear language to be used when creating the data object and it is quite easy in the above example, even for a non-technical person, to be able to read and decipher the majority of the operations that are occurring within the animation. This became a clear advantage in ensuring that non-technical designers could tweak and alter the animation description without requiring them to dig deep into the code that would ultimately animate the sprite on screen. The next step was to determine what information would be required to present to the designer in order for them to perform the desired actions such as moving, scaling, fading and rotating an image.

Structure of the XML Data Object

- Effect Type.
- Start Time.
- Duration.
- Start Value.
- End Value.
- Interpolator

This was the data structure ultimately decided on as it gave the most simplistic and clear layout with respect to what was going to happen within the animation. It would require very minimal technical knowledge for a person to read through one of these XML file descriptions and get a feel for what the animation would look like. This representation would be both immediately understandable by the designers and provide for

a very good basis from which to construct any additional features or implementations due to XML's previously mentioned rules.

Drawbacks of XML

There were some drawbacks in using XML and the most significant of these was the redundancy of readability once the user interface (UI) began to take shape. XML was suitable at first as without an interface to design animations the designers needed to be able to understand the animation file. However, once an interface was in place to sit between the design and end file then the readability of the XML file did not matter as much. Dropping the human-readability of XML for a compressed format would have enabled the system to shave slightly more information from the animation file and save space.

3.3 User Interface

While it was still unclear at the design stage of the project whether there would be time to incorporate a user interface for the designers it was something that seemed to push the end program to a higher level. As a result of this there was some initial design of the interface from the start in order to get a feel for what might be required in the future and thus the development would take a natural transition from the XML based system into a more visual and malleable representation of the animation. As well as this, having a user interface prevents easy to miss errors from cropping up in the creation of XML files, for example if the closing tag on an element is forgotten then it is not until the animation fails to run as expected within the virtual machine that this mistake is found and even then the designers will have to take time out to find the cause of the problem. With a GUI the XML has been constructed from within the user interface and so it will have been built correctly and the animation will play out as the designer has seen within the interface.

3.3.1 Initial Design

Views. The views of the concept interface (Figure 3.1) were planned to offer the user 4 different options, the standard top, front and side views which would allow the

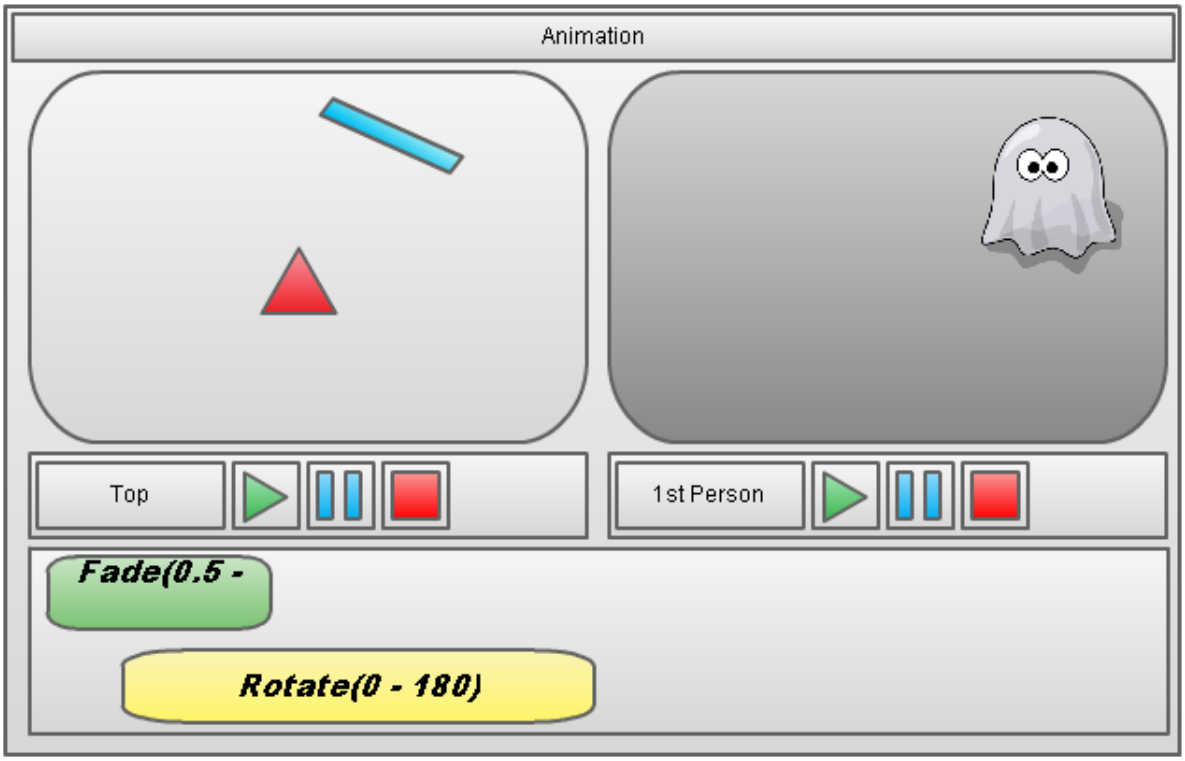

Figure 3.1: Initial GUI design showing dual views with playback controls. Timeline at the bottom shows duration and type of effects.

designer to see the scene around the player from a number of different angles and in addition to these three would be a replay view. The planned replay view would allow the designer to loop playback of the current animation which would update as the animation was updated, this view would also support play, pause and stop functionality.

Timeline. The timeline of the interface would be the region that would immediately show the designer the full flow of effects for the current animation. The time at which each of the effects began and the duration of each. From the timeline the designer would be able to click on any of the effects and be presented with a dialog box enabling them to alter the current settings. New effects would be added to the timeline by right-clicking on it presenting the user with a menu for selecting the type of effect that they would like to add.

3.3.2 Relative Positional Movement of Sprite to Player

As the game is set in the real world and the images within the game are going to be drawn on top of the players view through the camera it was important to the team that they were able to convince the player that the apparitions they were seeing were surrounding them. In order to achieve this it was important that objects were given positional information in such a way that they could be anchored to certain points within the real world while at the same time having a 3D representation. Having this sort of representation would allow for designers to create ghosts that flew out of windows, dropped down behind the player, or circled the player, creating more variety in the way ghosts were presented to the player and allowing harder challenges when attempting to capture the ghosts.

The spatial positioning was accomplished by taking the Z-axis of the Cartesian coordinate system (X, Y, Z) and using that to represent a player looking directly north in the cardinal direction system (North, South, East, West). The cardinal direction of the player is taken from a compass value within the Android device and so the image that is to be drawn can then be placed in the appropriate location within the real world and as the player moves around the world or turns to point their camera in another direction the 3D representation can account for this and the sprite will move accordingly. This allows for both anchored sprites and sprites that rotate around the

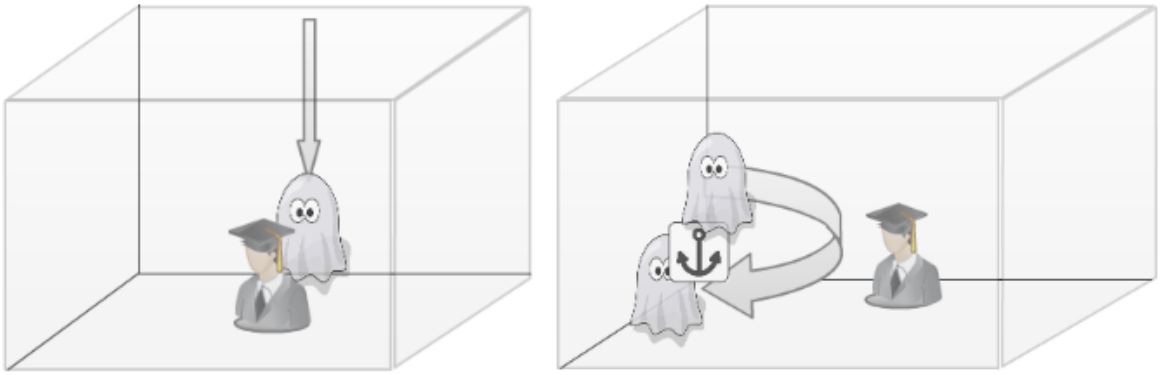

Figure 3.2: Concept drawings for the spatial positioning. The 3D world around the player can be used to position ghosts as required. Or the ghost can move based on an anchor within the world.

player or seem to originate from a particular point in the world before approaching the player as seen in Figure 3.2. This is represented in the animation designer interface (Section 4.4) with a red triangle in the center of the screen; this triangle shows the position of the player as they are facing north. Using this visual prop the designer can move the animated image, represented by the blue rectangle, in which ever way they please.

3.4 Design Summary

Having followed the requirements laid out by Haunted Planet the design decisions were based around the needs of the team. XML data representations would allow for fast prototyping of the animation framework and allow designers to have an early insight into how the system would work and any changes they would like to suggest. On top of this XML has a very small memory footprint within Android as there are already libraries and folders in place to deal with XML files. A graphical interface would certainly serve to blur the line between technical and non-technical users and even a conceptual premise would allow the initial implementation to favour a user interface friendly design. Spatial positioning within the real-world had been simplified and if the user interface meet the concept then designers would have minimal fuss creating animations that belonged to the real-world.

Chapter 4

Implementation

Once the general design of the system had been thought about and considered it was time to move onto the concrete implementation. While some items such as the proof of concept were run in parallel with the design process the majority of implementation was completed after the overall design of the system had been prepared. In this chapter there will be discussion on the tools and libraries used to implement the planned design and as well as this there will be a more in-depth look into the individual classes and components within the framework and how they all tie together. This will include a look at how the animation system pairs up with the OpenGL drawing functions and then onto the ways in which the designers shall interact with the final desktop tool. Some small implementation issues will be mentioned later in the chapter and the resolutions to these issues.

4.1 Proof of Concept for OpenGL Over Camera Stream

As there was no OpenGL rendering implementation for VGH at the beginning of the project most of the initial time spent with implementation began with the basic proof of concept. This included working out how the OpenGL layer interacted with the Android activity and just how much the typical Android device was capable of. It was also important to ensure that it would be possible to render an OpenGL 3D scene on top of the camera's rendering.

There was very little information about this available at the time but there were some Android examples for the camera callback functionality and one person who had had some success using this callback with OpenGL. When trying to investigate the basic camera rendering it was the first time that the fragmentation issues between the various Android devices and version histories would occur. Unfortunately the phone with which I was trying to test the concept was unable to run any Android platform past version 1.5 and as it would happen there was a problem with resizing the camera information in versions prior to 2.0 and so I could not compress the cameras information into manageable slices.

While I was able to get the camera stream displaying onto the screen, this was running at a stuttering 5-10FPS and this was before there were any additional images or computations being performed for the game. Fortunately the proof of concept was enough and the VGH team would be responsible for looking into how this problem could be resolved and in the end only devices running 2.1 or higher would be supported and so this solved the majority of camera issues. As the team began to develop a JNI implementation for the renderer this sped up the processing of the camera stream and any subsequent drawings on top of it.

4.2 Development Environment

When developing the framework for animation and particles there were a few different libraries and Integrated Development Environments (IDEs) that were used in order to get the best interface and easiest implementation for the job at hand. Google has focused its IDE implementation on the Open-source Eclipse IDE[13] and so this is the environment that was used to develop any code that would be run on the Android emulator or device. While the interface of Eclipse can generally regarded as less 'clean' and user-friendly than that of alternative IDEs it was the easiest to get the Android Emulator setup on as Google have easy-to-follow instructions on their site for how to do so and also aided any troubleshooting as it eliminated any rogue IDE conflicts as possible problems. Where possible, the emulator that was used to run any tests did so with a simulated 1.5 platform version of Android, this was to ensure maximum backwards compatibility with all devices. The only time when this rule was broken was to test the effectiveness of features only available to later releases of the platform.

While Eclipse sufficed for the majority of the initial development and testing on the Android device, once it was clear that a GUI was going to be the best route to take for designers it also became apparent that Eclipse was no longer the best environment for development. Unfortunately, Eclipse does not come pre-loaded with a GUI builder and due to the open-source nature of the project any external GUI builders can typically have unreliable and awkward interfaces. Thus it was decided at this stage to switch to the more clean IDE of Netbeans which is developed and maintained by Oracle (previously Sun Microsystems). Since Netbeans[42] has been developed by a single entity its design is much more fluid and well integrated, as well as this it has a built in GUI builder which is extremely intuitive to use.

Given that Java applications typically use the AWT and Swing libraries to generate GUIs and visual output for users this is what the default Netbeans applications were offering. In order to try and minimise the changes between the device output and desktop application output it was almost essential that the display within the application was performed using an OpenGL interface. There are a few different options in terms of gaining access to the OpenGL API within Java applications and thankfully the Java OpenGL (JOGL) implementation offered both a seamless integration with Swing and OpenGL but also extended the Netbeans projects wizard in order to accommodate new JOGL application projects[28][41].

In the end, the main bulk of the framework was transferred and written within the Netbeans IDE due to it's cleaner interface and more standardised methods for performing common tasks, while only Android specific code was written within Eclipse in order to be tested on the emulator.

4.3 Class Overview

4.3.1 Animation Effects

The most fundamental elements of the animation system are the effects that can be deployed by the designers in order to vary the behaviour of the image on screen. As mentioned previously the four effects available to the users are; translation, rotation, scaling and fading. Given the very close behaviour of each of these effects it was natural to work from a base class in order to implement each of the specific effect

implementations.

The base effect class contained a very strong representation of the functionality required by each of the specific effects and ultimately the concrete implementations for scaling, rotating and fading only required constructors that would set their type appropriately so that they could be easily differentiated later on in the code. Translation effects required more alterations to the base *Effect* class; mainly because the translation effect centered around the use of vectors for the calculations and not a single float value.

Each Effect had a *getValueAtTime(long time)* method which could be used to determine the appropriate interpolated value of an effect after a given amount of time. Individual effects were added into a list of objects within the *Animation* class which was then responsible for updating and averaging the individual effects. Finally, the *Animation* class then made a *SpriteState* object publicly available which represented all of the essential data to be used by the VGH application developers. The *SpriteState* class had variables for an animations current position, scale, etc... and these mapped directly to the OpenGL functions available.

glTranslate (float x, float y, float z) glTranslate is responsible for moving the draw pointer within OpenGL to new locations within the 3D world. As such it can be used to draw objects at different positions. This is the function which will be used in order to map the position details of an animation to on-screen movement.

glScale (float x, float y, float z) glScale offers developers a means of scaling any subsequent drawings that are performed within the OpenGL window. The scaling can be performed non uniformly along the 3 different axes which enables a skewing effect resulting in a "'stretched"' appearance of anything which is drawn. This is not used very often as the appearance of stretched objects is not very appealing. Typically for a 2D image the X and Y axes will be scaled uniformly in order to preserve the shape and consistency of the object. As the sprite is a 2D representation scaling the Z values will have no effect on the image itself.

glColor (float r, float g, float b, float a) glColor is a method which enables developers to taint the colour of objects drawn onto the display. For the purposes of testing the r,g,b values were kept at 1.0 which maintains the colour described by the texture

which has been applied to the image. The final *a* value, or alpha, represents the translucency of the drawing and so this value directly maps to the fading provided by animations.

glRotate (float x, float y, float z, float r) glRotate is used to rotate any subsequent drawings onto the graphics device around any arbitrary axis. The axis around which the rotation will take place is defined by the x, y, z components of the rotation method header. For example, an axis of (0, 1, 0) would cause any drawings to be rotated around the y axis, as if a person was turning on the spot. The magnitude of which the objects are rotate around the axis is specified by the final *r* parameter which represents a degree value.

4.3.2 Particle System

The particle system within the framework is based on Reeves' previously mentioned paper along with more recent developments[49][43] and so is not an overly complex system. The foundation of the system is a *ParticleEffect*; a particle effect is responsible for managing all of the individual particles involved in the effect, the location of the effect and the continuous updating or drawing of the effect.

To save on memory duplication there is a static random number generator initialised within the *ParticleEffect* class for any effect or individual particle to use. This saves some space as random numbers are used very frequently throughout particle systems and the number generator did not need a specific seed and so having a single instance was enough and saved space over each effect or each particle having their own generator. More significantly in terms of gain, a static generator will speed up the processing of particles.

Each particle effect then has a collection of settings which determine the behaviour of the particle effect. These settings are modified by the designer in the particle designer application in order to achieve the effect that they are looking for. The list that follows shows the variables that are being used to create particle effects in the particle designer. The results of the design are then output to an XML file to be later loaded onto the Android device and played back within the game as the designer has envisaged. When the particle effect is loaded onto the device for use in the game there is a similar interface between the particle system and the VGH game as there was with animations. First

the developer must create the particle effect through the object constructor, specifying the appropriate XML file and then the developer must update the particle effect for each update frame. After each update the state of each particle can be queried by the developer and so can then be drawn as required.

Particle Count. Number of individual particles within the effect.

Scale. Size of each of the individual particles.

Speed. Speed at which the particles will travel.

Direction. Determines the initial velocity of particles. In the range of 0-100. 0 - straight up, 25 - right, and continuing clockwise.

Spread. The angle of deviation from the given direction. For example, with a direction of 0 and a spread of 0, all particles will travel straight up. With a spread of 360 the particles will travel in a unit circle, regardless of the given direction.

Spin. The speed at which particles will spin.

Displacement X. A displacement along the x-axis from the particle effects initial position.

Displacement Y. A displacement along the y-axis from the particle effects initial position.

The values for these numbers are taken from slider bars within the designer. Since these slider bars range from 0-100 some modifications are applied within the particle settings class once the values are set. In particular, the angular values such as those for the direction or spread are taken as they are from between 0 and 100 and are then modified in order to comply with the 0-360 degree value expected by OpenGL. This is done with the simple equation *(sliderValue / 100.0f) * 360.0f.*

Once the value for the settings have been computed and passed down into the particle effect, the effect can then begin to generate individual particles. When a particle is created it retains a pointer back to its parent effect for ease of reference and is then reset. A particle can typically be reset for 1 of 3 reasons; the particle has just

been created, its parent effect has been reset or the particle has died as a result of fading out.

When a particle is reset it affects only some of its data; this serves to eliminate unnecessary calculations or adjustments and generally speed up execution. Specifically, when a particle is reset its position is returned to that of the parent particle effect emitter, taking into account any displacement that may have been specified by the designer. Along with this the particle is given new alpha, scale and rotation values to stop the particles looking too familiar to its predecessor. Directional calculations are omitted due to the number of more complex vector and angular mathematics involved. Using this particle representation which allows for easy alteration has allowed for a number of different effects to be created, from simple finger-tracking visual feedback or even basic weather or environmental effects such as rain, frost or snow can be simulated with ease.

4.3.3 SpriteBatch

Since the VGH JNI renderer had yet to be finalised it was necessary that visual testing for the framework could still go ahead and so a sprite batch class was created for the purpose of allowing test cases to be displayed on the Android device and also in order to encapsulate the drawing functionality into a single object. Ensuring that all drawing went through the spritebatch allowed for a number of benefits; first for the fact that any changes that needed to be made to how an was drawn could be done so easily and quickly through the spritebatch and secondly; any problems that were occurring when drawing objects could be pinpointed much faster as there was only a single route to take when drawing anything onto the display.

Following is the method used for drawing any image created by the framework. Since the images are represented by the state data structure of *(Position, Scale, Rotation, Alpha)* this is all that needs to be passed to the sprite batch in order for it to draw the image accordingly. Calling *glPushMatrix* at the top of the draw call preserves the world matrix state of any previous calls to the OpenGL device. To give an example of this, let us imagine that the player has turned the phone and now instead of facing north they are facing south; in order to accommodate for this the VGH application developer will have rotated the world by 180 degrees so that now the player is looking

at anything that is being rendered on the 'southern' side of the world.

If we did not save the state of this world matrix and push it onto the matrix stack, then the operations that we will be performing in the draw routine might affect that which has already been performed by the application developer. Once the matrix has been pushed onto the stack we can begin to transform the world ourselves by rotating it etc... but these operations are still altering a matrix unexpectedly for the application developer.

In order to prevent this from happening there is an additional step at the end of the draw method where the matrix that was pushed onto the stack at the start of the call is then popped back into the device from the stack with a call to *glPopMatrix*. This ensures that our draw was called with any transformations applied by the application developer previously and any subsequent transformations that we need to call are then applied on top of these. Once we have completed our translations and drawn our object we restore the world state to that which it was previously and so the application developer will not experience any unexpected changes.

Another benefit of having the encapsulated spritebatch class is that it was much simpler to vary the method used to draw objects with the OpenGL device. For instance, direct vertex arrays can be used to draw objects onto the screen but the most widely suggested method was to incorporate Vertex Buffer Objects and we will look at what they represent in the next section. The ability to switch the drawing method so easily will also be incorporated at a later stage in order to test the effectiveness of the various drawing approaches.

```
public void draw(AnimationVGH s)
{
 gl.glPushMatrix();

 AnimationState state = s.getCurrentState();
 gl.glTranslatef(state.getPosition().x,...
    ...state.getPosition().y,...
    ...state.getPosition().z);
 gl.glRotatef(state.getRotation(), 0, 0, -1.0f);
 gl.glScalef(state.getScale(), state.getScale(),...
    ...state.getScale());
```

```
 gl.glColor4f(1, 1, 1, state.getAlpha());

 ...OpenGL Interface Testing...

 gl.glBindBuffer(gl.GL_ARRAY_BUFFER, vboNames[0]);
 gl.glEnableClientState(gl.GL_VERTEX_ARRAY);
 gl.glVertexPointer(3, gl.GL_FLOAT, 0, 0);

 gl.glBindBuffer(gl.GL_ARRAY_BUFFER, vboNames[1]);
 gl.glEnableClientState(gl.GL_TEXTURE_COORD_ARRAY);
 gl.glTexCoordPointer(2, gl.GL_FLOAT, 0, 0);

 gl.glBindBuffer(gl.GL_ELEMENT_ARRAY_BUFFER, vboNames[2]);

 gl.glDrawElements(gl.GL_TRIANGLES, indices.length,...
    ...gl.GL_UNSIGNED_SHORT, 0);

 gl.glBindBuffer(gl.GL_ARRAY_BUFFER, 0);
 gl.glBindBuffer(gl.GL_ELEMENT_ARRAY_BUFFER, 0);

 gl.glPopMatrix();
}
```

Vertex Buffer Objects

Vertex Buffer Objects (VBO) are the suggested method of drawing objects for one main reason, the buffers which are used when drawing with VBOs are created on dedicated graphics memory[45][30]. Typically with vertex arrays you create the buffers or arrays in the application memory, fill it with the data that you want to push onto the graphics device and then send these buffers to the graphics device. This is slow as memory bandwidth is a bottleneck and this is particularly true for mobile devices. This process of filling the buffer with data and then passing them to the graphics device is repeated every frame or at least when the objects state has changed and so the data

needs to be updated.

VBOs offer a distinct advantage in that buffers created with VBO create a permanent memory allotment within the graphics device memory and so when the device needs this information it does not need to get it from the application memory, instead the information should be available in the much faster graphics memory cache. The application can then send new information directly into these VBOs and skip the lengthy process of copying buffer data from the application onto the graphics device.

There are three basic steps that need to be followed when creating vertex buffer objects. The buffers must be declared, filled with data and then bound before being drawn. Taking the code below as an example, the first step taken was to create a new buffer within the application that would hold the vertex information before it was passed down to the graphics device. When allocating the memory the buffer is given a capacity of *vertices.length * 4*, this is due to the fact that the vertices are represented by floating point numbers, each of which is represented by 4 bytes in total.

Once this has been completed the buffers are declared on the graphics device and essentially given names which can be used to later refer back to the buffer which has been created. This can be seen in the VBO code example with the call *gl.glGenBuffers(3, vboNames, 0)*. This function is essentially telling the graphics device that we would like 3 vertex buffers to be created and to store the names in the given array, in this case *vboNames*.

Finally the buffer is bound using *gl.glBindBuffer(gl.GLARRAY_BUFFER, vboNames[0])*. *GL_ARRAY_BUFFER* tells the device that the data that it will be receiving is going to represent vertex information and not indices or texture coordinates, *vboNames[0]* simply fetches the name of the buffer so both the application and the graphics device know which array is being referred to. From here the data can be passed down to the graphics device using the *glBufferData* command. Now the buffers have been declared, initialised and filled with the initial sets of information. The drawing is carried out in the same way as for regular vertex arrays, by calling *glDrawElements()*, only the vertex and index buffers are bound to those on the graphics device.

```
private void initialiseVBOs()
{
    ByteBuffer byteBuf = ...
        ...ByteBuffer.allocateDirect(vertices.length * 4);
```

```
    byteBuf.order(ByteOrder.nativeOrder());
    vertexBuffer = byteBuf.asFloatBuffer();
    vertexBuffer.put(vertices);
    vertexBuffer.position(0);
}

private static void setupVBOs()
{
    gl.glGenBuffers(3, vboNames, 0);
    gl.glBindBuffer(gl.GL_ARRAY_BUFFER, vboNames[0]);
    final int vertexSize = vertexBuffer.capacity() * 4;
    gl.glBufferData(gl.GL_ARRAY_BUFFER, vertexSize,...
        ...vertexBuffer, gl.GL_STATIC_DRAW);
}
```

4.4 Implementing the Animation Interface

4.4.1 Overview of Animation Interface

The final interface (Figure 4.1) which was able to be implemented was not very far in terms of features from the initial concept of what was desired. The views have been cut down and only offer the single top-down view but this was deemed to be the most important view as it accounts for the space around the player. This space around the player and the space that corresponds to the compass directions of the player were the most significant in terms of designing the positioning and animation of any images. This top-down view is the most useful as it is the general spatial area that the team were looking for i.e. moving the sprite around a player or ensuring that is it on a certain side.

There was another view planned that unfortunately was unable to be implemented in the time frame available. This view was a player's eye view (Fig.4.2) and would have given a vertical reference for the animation's position. As seen in the figure, the central red line indicates the center view of somebody looking directly to the north. The darker red lines on either side of this central line would represent the east and

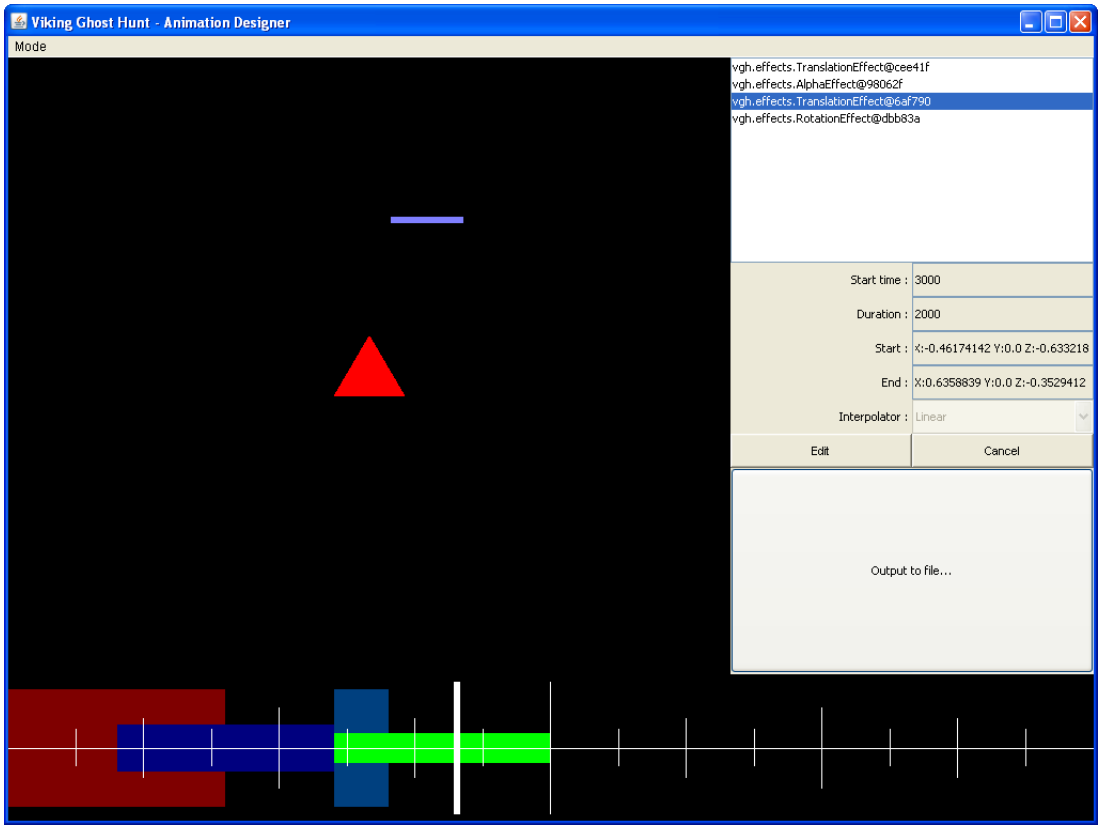

Figure 4.1: Overview of animation GUI.

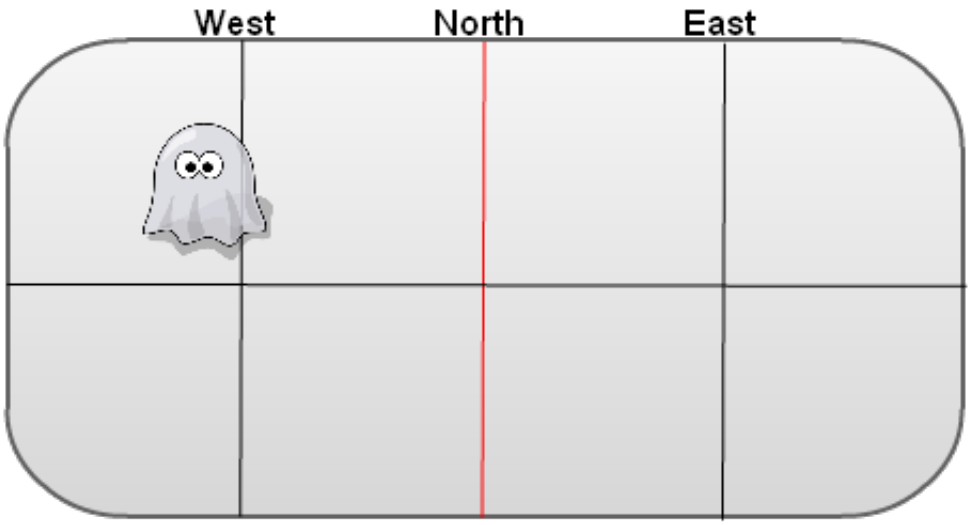

Figure 4.2: Sketch of planned player-view.

west extremities. It was the goal to have two separate views within the designer so that both the top-down view and the player-view could have been open simultaneously allowing for a larger degree of flexibility for the designer.

The effects are now listed in a listbox on the top-right side of the interface which allows the user to highlight effects that they have already added to the current animation. Below the effects listbox is the effect details which displays the information for the currently selected effect. This then allows for very easy access for the designer to alter the effects that they have already employed. Finally, below this is the save button which will bring up a saving dialog for the designer in order for them to save their animation to an external XML file.

The timeline at the bottom shows the desired visualisation of when each effect starts, its type, and how long its duration is. The effects can also be selected on this timeline and then edited by the designer. Significantly in the latest iteration of the interface the user can scrub the timeline using the white bar on the timeline, scrubbing the timeline allows the designer to play through the animation at the rate they are comfortable with or alternatively they can also focus on a certain point of the animation.

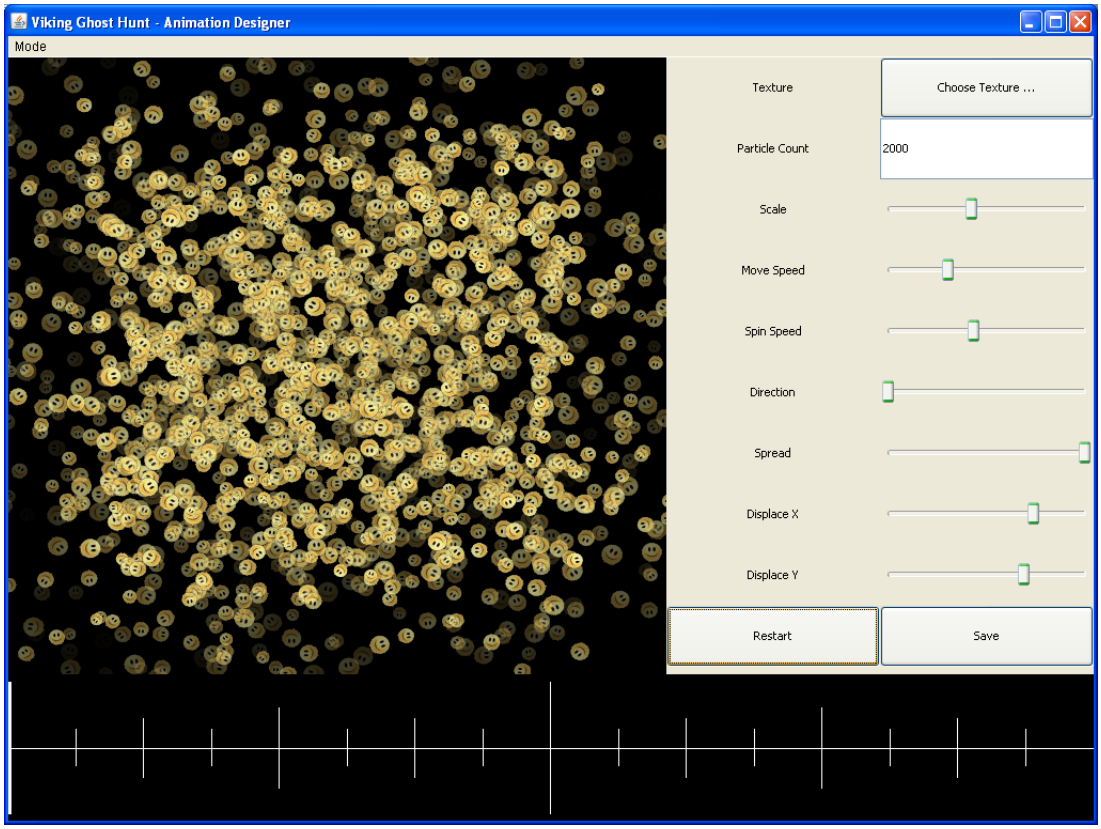

Figure 4.3: Overview of particle effect GUI.

4.4.2 Overview of Particle Interface

The interface for the particle system (Figure 4.3) is more simplistic as a lot of the complication for the animations are taken away. For instance, when switching to particle mode the timeline is no longer required and so this section of the interface is left blank. Any alterations are taking place on a single particle effect object and so there does not need to be a menu for selecting different effects as it is only a single effect. All that is left is are simple buttons for resetting the particle effect or for saving its output.

4.4.3 Components Within the Interface

View Implementations

The views within the interface have been developed using the JOGL OpenGL wrapper and are built up from a few layers of objects. The essential item is the OpenGL renderer which in the case of this implementation has been named *AnimationRenderer*. *AnimationRenderer* extends the *GLEventListener* interface which forces the renderer to implement the following three methods[29].

public void init(GLAutoDrawable drawable) The initialisation method for GLEventListeners is called once when the canvas has first been added into its container and so can be used by developers to create once off buffers or initialise any variables that may need to be instantiated. Most importantly perhaps is the drawable object that has been passed into the method; this object is our interface with the underlying OpenGL graphics device and so we can being to setup the configuration of the device and as well as that can now take the time to attach our extended *GLEventListener* interface onto the list of observers for the drawable object.

public void display(GLAutoDrawable drawable) The essential method within the implementation of a GLEventListener is the display method. This method is the one which will be called at every frame in order for the application to draw any objects required onto the display. Any operations for drawing the location and state of the player location or sprite will be described here.

public void reshape(GLAutoDrawable gLDrawable, int x, int y, int width, int height) Reshape is called when the viewport for the display has been changed, either the window has been shifted around the monitor or perhaps the user has opted to resize the application. This allows developers to account for uneven ratios of width:height and so can adjust the drawings accordingly so that they are not stretched.

Once the *AnimationRenderer* has been implemented the next step is to add this renderer into the UI itself and in order to do this the renderer must be placed within a container. Typically containers within a Java UI include things such as JFrames,

JPanels or similar; JOGL provides its own container for the renderer known as a *GLCanvas* which can be used to display a previously written renderer, in this case *AnimationRenderer*, which has been attached to the *GLCanvas*. This *GLCanvas* can now be placed into the hierarchy of a typical Java interface and for this project the *GLCanvas* was placed into the central position of the border layout for the main *JFrame*.

This view will now be responsible for displaying to the designers the current state of any animations that they are creating or any particle effects which they may be working on. Switching between the two different functionalities, animation and particle editing, is handled by a small boolean check which is changed when the user presses on the switch in the upper left hand corner of the UI.

The Animation Timeline

Given the importance of time within animations it is not surprising that timelines have been a staple of any recent animation software application. The timeline that was implemented for this project was designed to be kept clean and concise. Sitting at the bottom section of the designer it fills the window horizontally and displays any effects which the designer has already added. Each effect type has a corresponding colour code which allows the designer to quickly and easily see which effects are taking place at any given time and the time distance between them.

As well as this, overlapping effects are given vertical reductions in the space that they consume on the timeline in order to avoid obscuring any effects behind them. If any effect is selected from the listbox within the UI then the selected effect will be highlighted in green, again to give the designer a much better idea of where they are in the life of the animation and what it is that needs to change.

A scrub was added to the timeline class in order to give the designer the freedom to run through the animation at their own pace and stop at points that they felt might require tweaking. It also allows for a simple way to play the animation without needing to output the result and load it into the Android emulator in order to simply view the animation and determine if any tweaks are required.

The algorithms implemented for the scrub and timeline display are quite robust and allow for many effects to be added to an animation while all the while the timeline

will provide the designer a clear overview of the situation. The goal was to ensure that it was clear which effects were being used and at what time and it was essential that none of the effects were being obscured if they were both occurring within the same time frame or if there was any overlap.

First the timeslot class was kept to a minimal in order to encapsulate the information required without having any redundant information. In the end the timeslot class held four simple variables; the start time, duration, type of effect and a level. The first three variable are taken directly from the effect which has been generated by the designer, while the final variable, the level, is set by the *TimelineManager* with the algorithm shown later on and is responsible for the level or depth at which the effect will be drawn on the timeline. With this information the timeline could be constructed in such a way as to make it clear to the designer what was happening within the animation and when.

With the *EffectCollection* class giving the *TimelineManager* the current list of effects, sorted by start time and then duration, this is the psuedocode for the algorithm which allowed for an unobstructed display of all effects.

```
Timeslot[] openlist;
TimeSlot slot, highestOpenSlot, lowestFinishedSlot;

setLevels()
{
   For each timeslot in the sorted list
      If the openList is empty
         Add timeslot to open list and set level to 0.
      Else
         Find the highest level open slot in the open list.
         If we are completely inside that slot
            Set this slots level to +1 of the slot we are occupying
         Else If we are only partially inside of it
            Look for the highest level slot whose end time...
               ...is greater than our start time.
            If a suitable slot cannot be found
               Put this slot as the highest level slot.
```

```
            Else If a match is found
               Put this slot at a level equal to the match.
               Remove match from open list.
            Else if not overlapping at all
               Add in the next item at level 0.
}
```

Once the slots have been given an appropriate level it is easy to draw the corresponding timeline without obscuring any of the effects within the animation. This is done within the draw method of the *TimelineCanvas* class, *TimelineCanvas* is another JOGL *GLCanvas* used to draw the timeline. We give a maximum height for the timeslot, generally about 0.8 (1.0 being the height of the canvas) and a minimum height, usually 0.2 and this remaining space is then divided up depending on the maximum level that the timeline manager has given to any particular slot. For example, if when determining the level of slots the highest level slot was ranked at 5, then the height difference between minimum and maximum (0.8 - 0.2) would be divided by 5 and this would give an even spread of heights between the various levels and again ensure an as uncluttered and clear display as possible for the designer. As well as this the timeslot is drawn with varying different colours depending on the type of effect.

4.4.4 Using the Animation Interface

Adding a New Effect

New effects are added to the effect collection very easily through the user interface. With the exception of translations which will be looked at in a moment, the designer can right click on any part of the viewport and will be given a choice of effects which they can add onto the current animation. The choices as might now be familiar are fading, scaling, rotating and translation. Picking any of these effects will then present the designer with a new dialog box (Figure 4.4) allowing them to specify the specific details of the effect such as when the effect should begin, the duration of the effect and the interpolation that is to be used for the effect.

Further to this the designer is now given the chance to enter the start-end value pair for the effect; scaling, rotating and fading all have a single floating point start-end

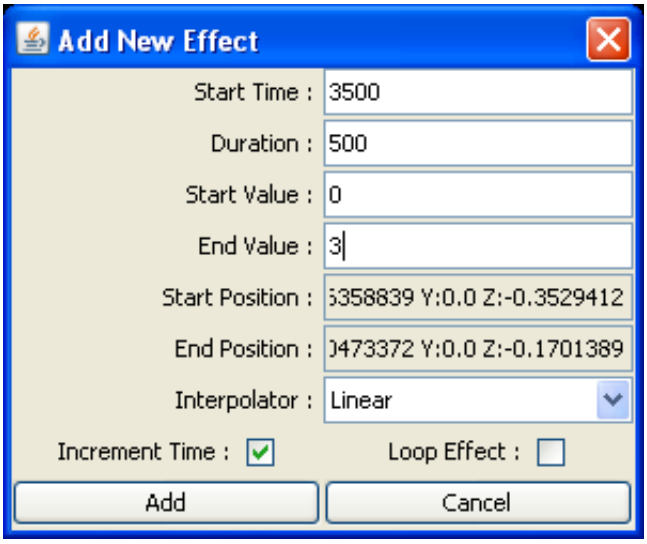

Figure 4.4: Adding a new effect to an animation.

set of values but the meaning of the values varies slightly. Scaling is quite straight forward in that a value of 5.0 will scale an object to 5x the size that it was originally where as a value of 0.5 will half this. Fading is matched to the coloring of OpenGL objects whereby a value of 0.0 is fully translucent while a value of 1.0 will result in full opacity.

Rotation is described in terms of degrees and so can be in the full range of the floating point number; values of negative rotation will cause a counterclockwise rotation while positive figures rotate the sprite clockwise. Interpolations will still be performed correctly with rotations outside of the -360 to +360 degrees of rotation so a designer can specify a rotation of 0.0 - 720.0 which will cause the sprite to rotate fully twice throughout the course of the rotation.

Describing Movement of a Sprite

In order for the designer to describe the movement of a sprite relative to the origin a similar process to adding the other effects is followed. First the designer must right-click in the view frame, illustrating the location at which the sprite should be when the translation is finished. From here the application automatically generates a starting position vector which represents the end point of the latest translation. These values are presented to the designer in the new effect dialog and the designer is then able

to tweak the typical values such as start time and duration as well as picking an interpolation for the translation. The designer can take this opportunity to tweak the vector positions if they wish or they can commit the new effect which will then be added to the list of effects and trigger the timeline to be readjusted.

When the translation effect is created there is a slight alteration in terms of how the translations work when compared to the other effects. This alteration is due to the fact that translations utilise vectors for the majority of their calculations. When the new translation effect is created a difference vector is calculated which essentially maps the linear path between the starting point and the end point. Each update then calculations the fraction of the difference vector that has been traversed. This fraction is then modified accordingly by the interpolator which has been selected by the designer and finally this interpolated vector is added to the start position to give the progress of the translation at the current time.

Editing Effects

Any effects that have already been added can later be edited if the designer has played through the animation and feels that it requires alterations. Selecting an effect from the list available within the UI will highlight the effect in the timeline to better illustrate for the designer which effect is being edited. Once an effect is selected the designer can press the edit button which will open up the textfields of the details section of the interface; with the text fields open the designer is free to adjust and tweak the values as they see fit.

Once the designer is happy with the changes they have made they can press the save button and the old effect will be updated with a new set of effect details which will have been extracted from the edited text fields. Once the old effect has been updated within the effect collection this will trigger an automatic refresh of the timeline so that the designer can immediately see the changes that they have made and they are then free to replay the animation again and determine whether or not the changes that they have made have brought the animation to a satisfactory level.

4.5 Consolidating Into a Single Framework

Once the initial design and implementation of the system had been complete it was then possible to take some time and look at how the framework would be used by the Viking Ghost Hunt team and what items were truly needed within the framework. This meant that is was critical that any unnecessary classes were removed from the distributable package in order to save the limited memory of the embedded device and it also meant that the interface between the framework and the end-users had to be as clean and uncomplicated as possible.

The stripping down of the framework resulted in a lot of classes that were used for testing being removed. These included classes such as the spritebatch and the general Android activity that had been used for testing. These classes were removed as the Viking Ghost Hunt team would already have a framework and methods in place for rendering textures to the screen and they had already begun to implement an OpenGL renderer, in fact, they had gone as far as to outsource a partial JNI implementation of the OpenGL rendering and so it would have been a much larger complexity to include OpenGL code in the final framework.

The interfaces between the two major components (animations and particles) were kept as clear as possible with as little dependency as possible. Ultimately the animation interface worked quite easily. Firstly the designer would be responsible for designing an animation with the animation designer application and once they were happy with what they had designed they could output this into an XML file within the Viking Ghost Hunt's XML folder. These files would then be automatically assigned a resource number by Android so that the XML file can easily be read into the Android application. Thus, the programmer responsible can load the animation simply by instantiating a new animation object with the XML file as its creation instructions. Once this has been done the animation is now ready to be played. From here the programmer can trigger the playing of the animation at any point that they choose, for example when the player has discovered an important clue, or perhaps once they have reached the location that they were walking to.

Once the animation is playing, it is not automatically drawn to screen, instead, the animation is simply updated along with the main update loop of the game and the state details within the animation are changed. These values can be queried within the

update loop or within a separate drawing loop and it is then up to the programmer how they wish to interpret these values. Most likely they will simply match the values directly to the OpenGL equivalent calls as had been done when testing the framework, for instance, glTranslatef(x, y, z) will be used to shift to the current position of the animation and then the JNI implementation will be used in order to complete the rendering of the ghost, object or character to screen. Using this abstracted method the animation can actually be used for more than simply moving ghost sprites around the world and may instead be used to implement a form of animated user interface.

Particles work in a similar fashion whereby the framework is not directly responsible for the drawing of the particles. The information which has been correlated in terms of which method is the fastest for drawing the particles has been forwarded to the Viking Ghost Hunt team and there is the option of drawing the particles with the best implementation found in the evaluation however this is disabled by default and instead the particle effect is updated similarly to the animation where it is updated within the games update loop and from here the Viking Ghost Hunt team can receive the position, scale, alpha and rotation of each of the particles and then can choose to render the particles as they like. Again, this allows for the drawing implementation to be flexible and allow for any optimisations that may have occurred when writing the JNI renderer and also allows for the fact that future versions of the Android platform may allow for faster rendering implementations and so the game is not locked down to any specific choice.

4.5.1 Stub! Using Android's Library Outside of Dalvik.

There were some hiccups when trying to ensure that the code running on the desktop application was as similar as possible to that running on the Android device and this mostly came down to a single complication. Due to the dex file compilation and that fact that the android libraries are aimed at the Dalvik VM, it is not possible to run the exact same libraries outside of the Dalvik environment. While this was not a huge complication for most things it did generate some issues every now and again. The most significant of these was the fact that the Android interpolators were unable to be used within the desktop application as they were simply stubbed from being used outside of the Dalvik VM.

This resulted in the unfortunate side effect of having the playback on the desktop application only show a linear interpolation of the animation as there was not enough time to work out the formula used for the android interpolations but this was deemed a minor sacrifice to make and the interpolators still perform perfectly within the Dalvik VM and provide more variety to the animation designers overall. There is a quick check done when the framework is used to test if the current VM can make use of the android interpolators and if it can then they are used but otherwise the interpolations fall back to the basic linear interpolation.

With more time it would be possible to investigate the formula that were used for the Android interpolators as the Android OS is an open source standard. This would allow for a complete 1:1 mapping of animation descriptions within the desktop application to that that is seen in the final execution on the device.

The stubbing of methods also affected the means by which textures could be loaded onto sprites and particles but this was not as significant a problem as this was just code used for testing and would not be used in the final distribution to the VGH team.

4.5.2 Wrapping OpenGL Objects

It was always the goal of the framework to share as much of the class logic and framework between both the designer and the final android application as this would cut down development time, debugging and aid the overall design of the framework in general. As a result of the designer application being developed for a desktop system while the animation or particles would run on the Android device, there were slight inconsistencies between the OpenGL interfaces that were being used on either platform. JOGL was the interface of choice for the desktop application in order to sync well with the UI design tools of NetBeans which come with a very well developed set of tools for JOGL. The android.kronos interface was used within the confines of the Android Device or Emulator and unfortunately as described in the previous section on Android's stubbing (4.5.1) of classes this interface was not easily available outside of the Dalvik VM.

The solution to this problem was to implement a bare-minimum wrapper for the two OpenGL interfaces and combine them into a single base class that could be specifically extended into either. This means that a base interface was created which would declare

all of the methods and constants that were being used within the implementation. Once this was established then the individual extensions of the base class could call the method implementation specific to either the JOGL interface or the Khronos one. This ensured that the majority of OpenGL code written in the framework would run on both the desktop application and the Android device/emulator without any change, the framework was just required to do a simple check at initialisation to determine which interface was being utilised. There were some aspects of the OpenGL implementation that were incompatible but these sections (such as loading textures) were used for testing and are not present in the final package of the particle and animation framework.

Chapter 5

Evaluation

As the framework would be executed on the slower hardware of a mobile device and would be required to support the minimum refresh rate for interactive applications it was important that the implementation was profiled and evaluated in order to ensure that it was executing as fast as possible or that no awkward loops were obstrucing progress. Here we will discuss some of the measures than to ensure that the framework would be suitable for the gaming environment and also test the limits of the particle system under different drawing methods which could aid the developers in final drawing implementation.

5.1 Maximising Performance

Once the major elements of the application had be written and tested it was time to look into optimising, tweaking and refactoring the code where required in order to get as much performance as possible out of the application. Performance analysis and testing is handled well within the Eclipse IDE as Android comes with a well equipped set of tools to deal with this. Firstly there is an easy logging tool which allows developers to print a selection of debugging or error messages to a log which can then be viewed as the program is running or looked at once execution has stopped.

5.1.1 Logging from Android

The logging tool is a static class and so can be accessed from within any of the classes within the project once the Logging library is included and the developer then simply calls one of five different methods within the *Log* class[20]. These different methods indicate the verbosity of the message and thus serve to indicate severity. The five methods are *e, w, i, d, v* and correspond to error, warning, information, debug and verbose messages. When the program is ultimately compiled error, warning and info logs are kept, however it is recommended that the developer ensures that any verbose messages are removed from the program.

Using the logging system, it was possible to track problems within the program as it executed but in terms of evaluating performance the log was mostly used for measuring the frames per second of the application and generally tracking the program's execution progress. Frames per second (FPS) represents the number of times each second that the display is being redrawn and is typically the standard measure of performance for graphical applications. A good range of FPS for handheld devices is typically considered to be about 30 as the action is generally never too intense that the refresh needs to be any faster than this. Fast games on more powerful desktop or console machines will generally require a minimum framerate of 60FPS to be considered smooth.

5.1.2 Profiling and Optimisation Hazards

When looking to improve the quality and performance of the application there are a few points of note which Google has suggested that developers should keep in mind[22]. First and foremost are the two essential principles of software optimisation; 'Don't do work that you don't need to do.' and 'Don't allocate memory if you can avoid it.'. These principles are then followed by a few old adages that stay true even today, and perhaps the most pertinent of them is the following quote from Donald E. Knuth who said, 'Premature optimization is the root of all evil'.

This embodies the common thinking that a programmer should first worry about the overall architecture and soundness of their program before they begin to pick away at the minor elements within their program where they find the potential for some gain. Often times, programmers will seek out these optimisations without really knowing the

effect that they are having on the program overall as they are unaware of what is really taking up time within their execution and thus they are looking in the wrong place of the code to make optimisations.

With this in mind it was not until further down in the development of the framework, once the major structure had begun to make itself apparent and any architectural changes that needed to be adjusted for had taken effect that optimisation would be considered. Once this has been accomplished it was time to investigate the performance of the system in detail and determine if there were any points within the system that could be improved. As described earlier, the first step in optimisation was the find the bottlenecks of the program as it was executing. This is not too difficult to achieve for Android applications as the aforementioned profiling tools allow for clear and precise tracking and post-analysis of a program's execution.

A developer can start the android profiler at any point in the execution of a program and the information gathered from this profiling will be gathered in a trace file that will provide a host of information to the developer in locating the bottlenecks of a system. These files can then be analysed using Android's Traceview program[23] an example of which can be seen in Figure 5.1. In this example it is clear to see that *glDrawElements* (in pink) is consuming the vast majority of execution time and so it is then clear that drawing optimisations should be considered and the next section will cover that.

5.2 Particle System Feasibility

One of the larger areas of performance interest was the particle system. A single image animating on screen is not going to produce significant trouble for modern processors. However when we multiply the items being drawn to screen and then have upwards of 50 objects it starts to become a more complex problem. Following the general cycle for a game the particles within the system will need to update during each update frame of the game and depending on the number of particles in the system this can have a significant impact on performance.

During the update of a particle, as many redundant calculations as possible have been removed in order to ensure that as each update is required to be called it is only performing the calculations that are absolutely required. As such, for each particle there are a few operations which must be carried out at the beginning of the particle's

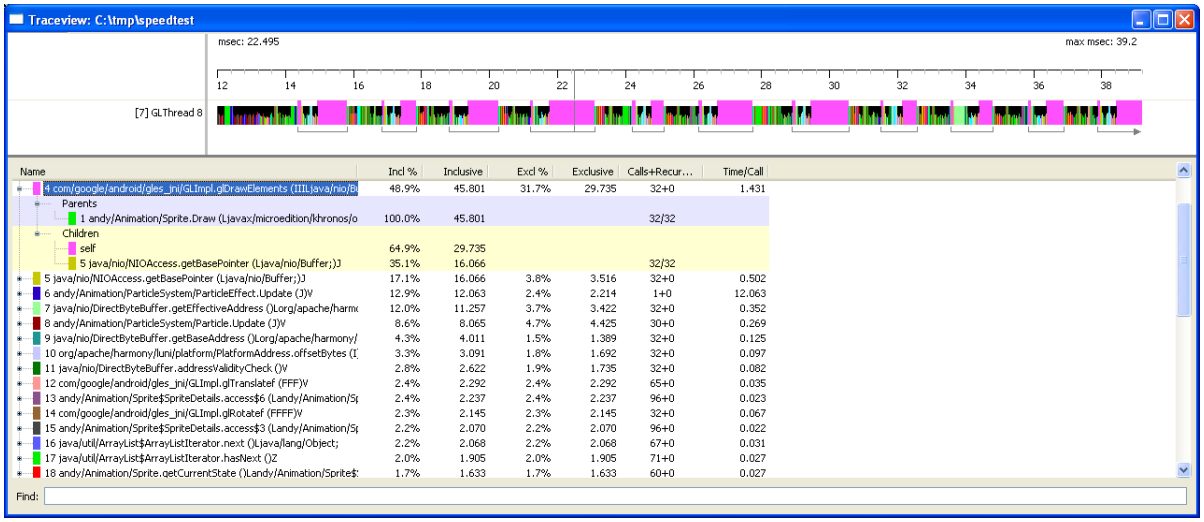

Figure 5.1: Profile of particle system using Traceview. Pink regions within the graph indicate calls to *glDrawElements*.

life and so spare time calculating in the update loop. Within the update loop the particle must have its position updated, its lifetime reduced and any alternative operations applied such as spin to the particle. Minimising the update requirements can lead to significant gains as the update is called many times before the particle is reset.

Using the traceview profiler mentioned in the previous section when running particle effects reveled a very telling truth, the majority of time was being spent drawing the particles. Unfortunately the profiler cannot dig into the OpenGL code in more specifics, however the profiler could provide enough information to ensure that it was the drawing which was slowing things down. Having seen the telling revelations of the profiling it was a natural progressing to then begin to look into ways at making the draw calls for the particle system faster. While the drawing module would ultimately not be passed onto the team, any knowledge which could be garnered from the profile and aid the final team in implementing the particle system would be worthwhile knowledge.

Particles were first tested with a number of different methods for drawing the particles, this was achieved with minimal interfere with the code as the previously mentioned *Spritebatch* was modified for the separate methods. Firstly the particles would be drawn as individual single quads (A quad is a 4 vertex mesh generally composed of 2 separate triangles sharing common vertices.), one at a time. Following this the quads for the particle system would be grouped into a batched draw call so that the position

Particle Count	20	32	64	96	128
FPS	57	50	35	32	25

Table 5.1: Frame rates when drawing individual quads.

Particle Count	32	64	96	128	160
FPS	56	44	36	32	26

Table 5.2: Frame rates for a single draw batch.

and orientation of all of the quads would be calculated beforehand and then a single draw call to the OpenGL framework would be dispatched with large buffers in order to reduce the number of draw calls. Finally the OpenGL Extension Point_Sprites was used in order to determine how effective they would be.

Tests (shown in Tables 5.1, 5.2 and 5.3) revealed that point sprites were in fact the fastest way to render the particles and this should come as no surprise as the purpose of the Point_Sprite extension is to give hardware acceleration to simple textured quads. The problem with this however is that due to the extensible nature of OpenGL Point_Sprites may not be available on each of the handsets in the market. In addition to this, while the point sprite was available for the G1 device used to test the implementation, the texturing component was not and so all particle drawn had to be of solid colour.

With this in mind the decision then comes down to drawing batches of quads versus individual draw calls. The gap between individual draw calls and batch calls are not quite as significant as those of the point sprites but any gain on mobile phones should be considered. Most drawing pipelines will be easily able to commit and draw a single quad and so using individual draws will probably not take an awful lot of tweaking and restructuring of current implementations and so as an intermediate solution could be quite inviting. Batch calls will prove useful in many cases however and so it is something that should be aimed for if it is not possible immediately. There is some more pre-planning involved as buffer sizes need to be calculated in order to store all

Particle Count	32	320	1280	2560	3200
FPS	57	57	46	33	28

Table 5.3: Frame rates when using the Point_Sprite extension.

of the vertex information but this should not be a major deterrent for sacrificing the gains that can be made.

In addition to how the particles are drawn there are also a number of different factors with respect to how the OpenGL pipeline is setup which can affect the speed of drawing the particles. A great guide in determining what to look for when trying to increase the speed of rendering for mobile applications can be found in the book Mobile 3D Graphics[30] which includes a chapter on 'Performance and Scalability'. Using this chapter it is a much easier investigative process in determining what is slowing down the draw calls for an application and also which is having the largest effect. This can be something such as enabling depth testing with an orthogonal matrix, which is unnecessary, or it might be a suggestion that the viewport is reduced in order to see what is specifically limiting the drawing capabilities of the application.

5.3 Programming for Performance

Once a suitable section of code has been located through the use of the profiling analysis it is then time to figure out reasons why that particular might be performing slowly or perhaps ways in which the problem area might be able to perform faster through better use of environment specific improvements. A host of possible efficiency improvements have been documented by Google[22] in order for developers to speed up any trouble regions within their execution but the information also provides for a number of good practice items that should be used whenever possible.

Avoid Creating Objects This is one of the most fundamental suggestions provided by Google. As discussed already in this report, memory is restricted on embedded devices and when developers create objects, even just temporary objects, they do not come for free. These objects take up space and due to Android's garbage collection they will hang around for a while before finally being cleaned up. Accompany this with a games natural looping behaviour and creating objects within the loop can have large implications as you create a temporary object, use it, and then the game loops around again. On this second loop the previous object has yet to be garbage collected and so there is another object created to fill the same purpose as the discarded one.

This tip was a very suitable reminder for this project and was able to eliminate a number of temporary *Vector3* objects that had been created in an update loop in order to calculate new velocities. Instead of making these temporary objects a global object was created which could be used as required and if possible vector calculations were done directly on the object in question and not on an interim value.

5.4 Evaluation Conclusion

The evaluation of the framework code and subsequent refactoring was able to clean up both the clarity of the code and its execution. Using Android's profiling tools, bottlenecks were quickly highlighted which ensured that time was spent in the appropriate locations of the framework. With less objects being created in critical update loops this served to minimise waste that was having to be collected by the garbage collector and thus resulted in stuttering. The particle system information will provide useful test cases to the team when they incorporate particles into their rendering implmentation as they will know that there is potential for gain with certain approaches if they are not already using them or if they are indeed capable of using them.

Chapter 6

Conclusion

6.1 End Product

Ultimately, the framework and tools have been developed for the requirements set out by the client and for that the application has meet and in some cases exceeded the demands. Originally the request was for a simple text-based representation that would allow animators to describe animations but this never felt quite right and so the underlying representations were evolved into graphical implementations once the skeleton of the textual animation description and execution had been carried out. The designers for Viking Ghost Hunt now have a designer friendly visual tool which they can use to provide additional immersion for their players and will not suffer from the significant memory issues which they had been experiencing previously. While the development of the framework had to unfortunately be split too thin in some directions, between the OpenGL test suite, the user-interface for the designers and the animation framework itself, there has been a lot of progress made towards offering the designers of Haunted Planet Studios a variety of options and possibilities when developing story and art throughout the continuation of their development.

6.2 Improving the Development Process

6.2.1 Improved Communication

Due to the projects close ties to an outside team it was ultimately being developed for Haunted Planet Studios. Perhaps the largest downfall with the project was the fact that not enough communication had taken place between the tool development and the designers and team that would ultimately be using it. As the team was testing the game while the animation tool was being developed it was not the most fitting time and along with this there was an overhaul of the interface and style of the game throughout the development of the tool. This should have been worked around however as the tool would have gained invaluable benefit from more contact with the design and development team at Haunted Planet Studios whenever it was possible. There should have been closer contact with the designers allowing them to ask for functionality that they would like or how they would want the interface to be shaped but unfortunately it did not happen often enough.

6.2.2 Alternative Approach

Part of the trouble with the implementation of the animation system was that there was work required in terms of building an OpenGL implementation just for testing while the VGH team were completing the JNI implementation. An alternative approach to building the animation system would perhaps be to consider using one of the larger 3D design applications currently available for public use. This would not have to incur any new costs as there are a number of open-source and free to use 3D design tools available such as Blender[5] which would have more than enough flexibility to cater for the needs of the project. It might then be possible to write a plugin for one of these tools which would allow the designer to use the features already available within these design applications in order to describe the animation, the difficulty would then come from initially learning the framework and architecture of the 3D application and then being able to alter the tools provided by the application to design animations.

Finally the animations created within the application would then need to be pulled from the application and used within the VGH game. Since the implementations within the desktop application and device handset would be intrinsically different this may

require some additional work in ensuring that the animation that is specified by the designer is coming across as intended on the device. This should be very manageable given the time that would be saved in terms of not having to worry about drawing elements within the tool or creating the user interface for the designer.

The user interface for the applications might also be something that the designers are already familiar with and it would be worth while to survey the design team and determine if there was any tool that they had used previously which would be open to plugin support or general tinkering so that the designer would already have experience and feel comfortable using the application.

6.3 Future Work

While the tool was useful for developing the animations required for Viking Ghost Hunt there were some things that it lacked and other elements that could have been expanded on in order to improve it. In terms of improvements to the current system it would have been very advantageous to have the ability to switch the viewport in the animation designer or perhaps implement a 3D viewport. Switching between view ports would allow for the designer to look from a top-down view and from a first-person view in order to better design animations that occurred along the y-axis or the players up-down range. Further from this a fully realised 3D viewport would be ideal as it would allow the designer to specify clear 3D locations that they wish for the sprites to appear in and translate to but this would perhaps be an unnecessary complication for the designer to have to deal with as 3D viewports can be more complicated to deal with.

Improving the speed with which the particle system was able to draw its particles was also an area that could have used more investigation. While it was clear and evident that Vertex Buffer Objects were the best choice in terms of accelerating the draw speeds it was not evidently clear what method of loading the new particle data was best. Investigation into whether it is faster to load all vertex and texture information into the buffers before executing the draw or if it is equally as fast to do the writes and subsequent draws in sequence would be an interesting area to look into and could improve the impact of the particle system.

As a new addition to the framework it would be extremely useful to look into the

vertex and fragment shaders that are available to any Android phone running version 2.0 or later. Implementing shaders into the game would bring it to a whole new level and with the accessibility that the newer handsets offer it would be a missed opportunity to not utilise the processing power of the GPU in some respect in order to divert work away from the central processor. While it would not be a easy for a designer to incorporate this functionality as most of the advantages and techniques would require a high degree of technical knowledge, with a clean interface between the shaders and the underlying application of the shaders it would be enormously powerful if shaders could be hot-swapped onto images and would allow for near endless possibilities and effects.

Bibliography

[1] Dave Shreiner Aaftab Munshi, Dan Ginsburg. Opengl es 2.0 programming guide. 2009.

[2] AndroidAndMe. id software demos the power of smartphone gpus. http://androidandme.com/2010/08/news/id-software-demos-the-power-of-smartphone-gpus/, 2010.

[3] Apple. Apple app store. http://www.apple.com/iphone/apps-for-iphone/#heroOverview, 2010.

[4] Apple. Apple app store; paid apps chart. http://www.apple.com/itunes/charts/paid-apps/, 2010.

[5] Blender. Blender homepage. http://www.blender.org/, 2010.

[6] Falkland Ghost Hunt Blog. Homepage. http://www.hauntedplanet.com/, 2010.

[7] Dan Bornstein. Dalvik vm internals. http://sites.google.com/site/io/dalvik-vm-internals, 2008.

[8] Fast Company. The great app bubble. http://www.fastcompany.com/1684020/the-great-app-bubble, 2010.

[9] comScore. Smartphone adoption shifting dynamics of u.s. mobile gaming market. http://www.comscore.com/Press_Events/Press_Releases/2010/4/Smartphone_Adoption_Shifting_Dynamics_of_U.S._Mobile_Gaming_Market/(language)/eng-US, 2010.

[10] Distimo. Distimo monthly reports. http://www.distimo.com/report/, 2010.

[11] Entertainment and Leisure Software Publishers Association. Consolidated sales; 1998 to present. http://www.elspa.com/?i=3944, 2010.

[12] Forbes. Doom'ing the iphone. http://www.forbes.com/2008/07/24/doom-iphone-morris-tech-personal-cx_cm_0725doom.html, 2008.

[13] The Eclipse Foundation. Eclipse homepage. http://www.eclipse.org/, 2010.

[14] Robi Sen Frank Ableson, Charlie Collins. Unlocking android a developer's guide. 2009.

[15] GamesIndustry.biz. Jobs' game. http://www.gamesindustry.biz/articles/2010-09-03-jobs-game-editorial, 2010.

[16] Gartner. Gartner says worldwide mobile device sales grew 13.8 percent in second quarter of 2010, but competition drove prices down. http://www.gartner.com/it/page.jsp?id=1421013, 2010.

[17] Google. Androidology - part 1 of 3 - architecture overview. http://developer.android.com/videos/index.html#v=QBGfUs9mQYY, 2007.

[18] Google. Android developer homepage. http://developer.android.com/sdk/index.html, 2010.

[19] Google. Android homepage. http://www.android.com/, 2010.

[20] Google. Android log. http://developer.android.com/reference/android/util/Log.html, 2010.

[21] Google. Android market. http://www.android.com/market, 2010.

[22] Google. Designing for performance. http://developer.android.com/guide/practices/design/performance.html, 2010.

[23] Google. Traceview: A graphical log viewer. http://developer.android.com/guide/developing/tools/traceview.html, 2010.

[24] Kronos Group. Opengl extension specifications. http://www.khronos.org/registry/gles/, 2010.

[25] Kronos Group. Opengl homepage. http://www.opengl.org/, 2010.

[26] The NPD Group. Motorola, htc drive android to smartphone os lead in the u.s. http://www.npd.com/press/releases/press_100804.html, 2010.

[27] Viking Ghost Hunt. Information page. http://www.ndrc.ie/portfolio/entertainment/viking-ghost-hunt/, 2010.

[28] JogAmp. Jogl homepage. http://jogamp.org/, 2010.

[29] JOGL. Jogl documentation. http://jogamp.org/deployment/jogl-next/javadoc_public/, 2010.

[30] Ville Miettinen Kimmo Roimela Jani Vaarala Kari Pulli, Tomi Aarnio. Mobile 3d graphics with opengl es and m3g. 2008.

[31] Kotaku. Play with the unreal engine on your iphone with epic citadel. http://kotaku.com/5627701/play-with-the-unreal-engine-on-your-iphone-with-epic-citadel, 2010.

[32] Kotaku. Project sword is epic's first unreal engine iphone game. http://kotaku.com/5627603/project-sword-is-epics-first-unreal-engine-iphone-game, 2010.

[33] Aaron Brown Kurt Jung. Beginning lua programming. 2007.

[34] Lua. Lua homepage. http://www.lua.org/, 2010.

[35] TIME Magazine. 25 years of tetris: From russia with fun! http://www.time.com/time/arts/article/0,8599,1902950,00.html, 2010.

[36] Microsoft. Direxts homepage. http://www.microsoft.com/games/en-us/aboutgfw/pages/directx.aspx, 2010.

[37] Nintendo. Consolidated sales; 1998 to present. http://www.nintendo.co.jp/ir/library/historical_data/pdf/consolidated_sales_e0912.pdf, 2010.

[38] Nintendo. Gameboy homepage. http://www.gameboy.com/, 2010.

[39] Nokia. History of snake. http://conversations.nokia.com/2009/01/20/history-of-nokia-part-2-snake/, 2010.

[40] Oracle. Oracle files complaint against google for patent and copyright infringement. http://www.marketwatch.com/story/oracle-files-complaint-against-google-for-patent-and-copyright-infringement-reflink=MW_news_stmp, 2008.

[41] Oracle. Jogl for netbeans homepage. https://netbeans-opengl-pack.dev.java.net/, 2010.

[42] Oracle. Netbeans homepage. http://netbeans.org/, 2010.

[43] Kim Pallister. Game programming gems 5. 2007.

[44] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[45] Dave Shreiner. Opengl programming guide 7th edition. 2010.

[46] John T. Stasko. A practical language for software development. 1990.

[47] Haunted Planet Studios. Homepage. http://www.ndrc.ie/portfolio/entertainment/haunted-planet-studios/, 2010.

[48] TechCrunch. Gameloft made $25million from the app store last year. http://techcrunch.com/2010/02/02/gameloft-iphone-revenue/, 2010.

[49] Naty Hoffman Tomas Akenine-Moller, Eric Haines. Real-time rendering 3rd edition. 2008.

[50] VGCharts. Total worldwide sales per console. http://www.vgchartz.com/hardware_totals.php, 2010.

[51] W3C. Xml homepage. http://www.w3.org/XML/, 2010.

[52] Richard Williams. The animator's survival kit. 2001.