

# OLANdroid

## Final report for CS39440 Major Project

04 May 2015, Release 1.0

### Author

Gideon MW Jones  
gij2@aber.ac.uk

### Supervisor

Neal Snooke  
nns@aber.ac.uk

This report is submitted as partial fulfilment of a BSc degree in Computer Graphics, Vision and Games (G451)



Department of Computer Science, Aberystwyth University, Aberystwyth, Ceredigion, SY23 3DB, Wales, UK

Ethics application ID: 844



110068497

# Declaration of Originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature ..... (Gideon MW Jones)

Date .....

# Consent to Share this Work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature ..... (Gideon MW Jones)

Date .....

# Acknowledgements

I'd like to thank my family and my flatmates for supporting me and the sense of camaraderie respectively throughout this project and the whole of university. Without their support and fresh take on things, this project would not be as developed as it now is.

A further thanks go to the members of the beta testing group, who provided useful feedback and ideas for improving the project. These include Ollie Roe, Kit Farmer, Tom Rosier and Alex Stuart.

Also deserving of gratitude is my supervisor, Neal Snooke, for offering experience with the subject matter, analytical discussion and motivation from the beginning of the project.

# Abstract

Aerobatics – a portmanteau of aerial acrobatics – is the practice of flying manoeuvres not usually done in normal flight. These manoeuvres can be flown by full-sized aircraft and model ones alike and is often a means of recreation, entertainment and competition.

Learning these manoeuvres is a complex task, utilising a large amount of spatial visualisation and study of the diagrams. These methods do not provide a satisfactory way of building up an understanding of these manoeuvres, so a modern approach is proposed, making use of 3D graphic technology to build up a virtual, visual model of an aircraft's flight based on a user's input.

This project is centred around providing this dynamic visualisation of these acrobatic flights on a convenient platform – Android. It gives learners a catalogue of standard manoeuvres, allows them to compose their own flights using the OLAN system, and demonstrates the flight in progress through a model and animation.

This document details the research, analysis, design and implementation processes of developing the OLANdroid application.

# Table of Contents

1	Background.....	7
1.1	Aerobatic Flight.....	7
1.2	Project Task Definition.....	8
1.3	Motivation.....	9
1.4	Relevant Projects.....	10
1.5	Platform and Technology.....	10
2	Analysis.....	12
2.1	Representation of Manoeuvres.....	12
2.2	Flight Description Interpretation.....	14
2.3	Visual Modelling of a Flight.....	15
2.4	User Experience Focused Features.....	17
3	Process.....	18
3.1	Introduction.....	18
3.2	Key components.....	19
3.3	Preparation.....	19
3.4	Iteration.....	20
3.5	Supervision.....	21
3.6	Summary.....	21
4	Design.....	22
4.1	Introduction.....	22
4.2	Overall Architecture.....	23
4.3	Design in Detail.....	24
4.4	User Interface Design.....	33
5	Implementation.....	38
5.1	Code.....	38
5.2	Development Timeline.....	38
5.3	Finer Details.....	41
5.4	Issues.....	42
6	Testing.....	44
6.1	Device testing.....	44
6.2	Unit Tests.....	45
6.3	Stress Testing.....	46
6.4	User Testing.....	47
7	Critical Evaluation.....	49
7.1	Research and Analysis.....	49
7.2	Process.....	49
7.3	Design.....	51
7.4	Features and Extension.....	52

7.5	Platform and Tools.....	53
7.6	Conclusion.....	53
8	Annotated Bibliography.....	54
9	Appendices.....	57
9.1	Libraries and Tools.....	57
9.2	Glossary.....	57
9.3	Manoeuvre Catalogue XSD.....	59
9.4	Unit Testing table.....	59
9.5	Document history.....	59

# 1 Background

## 1.1 Aerobatic Flight

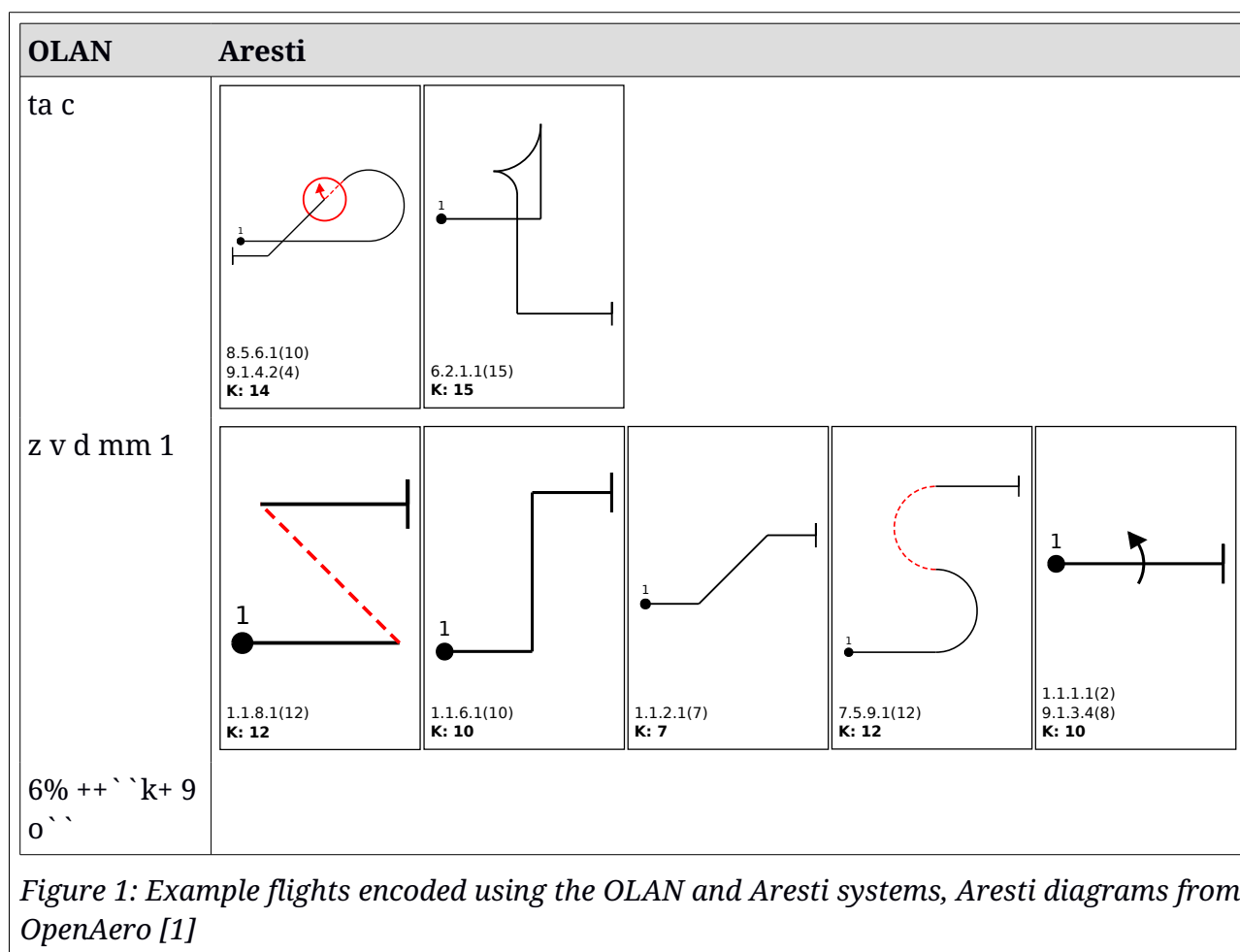
The acrobatic flight of aeroplanes is the practice of flying in patterns more extreme than those commonly used in flight. There's a number of different motivations behind this activity, including recreation, entertainment, sport and training. While primarily aeroplanes are used, various other flying vehicles can participate in aerobatics – gliders and helicopters – the latter with its own more specific set of movements available.

Full-sized aircraft can be used for aerobatics, but model aircraft – of the powered and controllable variety – are more commonly flown by hobbyists. They come in many different shapes and sizes, with different flight characteristics. The smaller nature of these aircraft means that forces exerted on them are far less significant than those on a full-sized plane, and that there's no worry about the in situ G-force a pilot would have to endure. As a result, far more intense flights can be undertaken with model aircraft than full-sized ones, though that doesn't rule them out from being able to perform as well.

An aerobatic flight can be broken down into manoeuvres, each representing a pattern of movement for the aircraft to follow. A commonly known example manoeuvre is a “loop-the-loop”, consisting of flying a plane round in a vertical loop. A wide range of these manoeuvres exists, describing many different patterns involving movements on every axis – pitch, yaw and roll – as well as even backwards movement (tailslides).

### 1.1.1 Notations

Manoeuvres can be described in two main different ways – the Aresti and OLAN notations (Figure 1).



The Aresti system was developed by Spanish aerobatic pilot José Louis Aresti Aguirre, and has been adopted by the Fédération Aéronautique Internationale (FAI) as a standard. It represents manoeuvres using simple diagrams and numbers (catalogue references). The prior is pretty simple to understand, but the latter relies on an official catalogue which is only available in a physical format for a price.

Alternatively, OLAN (One Letter Aerobatic Notation), developed by Michael Golan, provides a more concise and independent method of representing manoeuvres. It reduces manoeuvres instead to a simple letter description, with several modifiers available, such as extended entry/exit and scaling to allow for more complex flights.

## 1.2 Project Task Definition

In the context of the field of aerobatic flight, the focus of this project is the visual modelling of the acrobatic flight. The visualisation provided needs to be dynamic, being able to demonstrate various user-defined flights; and clear, it must effectively show the movement of the aircraft so they can develop an understanding of the flight.



The target audience for this project is acrobatic flight enthusiasts, particularly beginners. This focus adds concision and direction to the visualisation aspect, and opens up the project to features not necessarily directly tied to that main focus with a widened scope. This defined target audience suggests the development of a more complete, intuitive experience.

## 1.3 Motivation

Developing a 3D understanding of an aerobatic manoeuvre can be hard, involving a good degree of spatial visualisation thinking and imagining the way the aircraft moves. Beginners are often taught it by another person, with the aid of a stick plane, (essentially a tiny model plane on a stick you can move around to emulate manoeuvres), by studying Aresti diagrams and interpreting them or using resources like videos. None of these methods are perfect or ideal, with either requiring another person, an Aresti catalogue or finding relevant, reliable resources and interpreting them correctly.

Part of the work of this project was the encoding of these flights into the application, and thus it was important that there was an understanding of them – a first-hand experience of the problem this project aims to tackle. The supervisor was of some use, explaining some manoeuvres, but this was found to not be a great help. Videos of flights online and long studying Aresti diagrams (built in OpenAero, detailed later) were more helpful, but it still took a while to build a full, coherent picture of what the aircraft was doing. This reinforces the demand for such an application – this project aims to provide a functionality useful to people learning to fly these manoeuvres and build up a picture of how the plane moves without the facilities suggested.

### 1.3.1 Personal Motivation

On a personal level, there's an interest in all three of the main components of the application – aviation, 3D graphics and Android development.

From a youth spent living near to a major airport, with a lot of time playing flight simulators, and a love of Airfix model aircraft passed down from a father, an intrigue in aircraft was developed.

Multiple courses at university created an interest and skill in 3D rendering, with tuition and experience with the fairly high level Java3D and the much lower level OpenGL and related GLSL, building knowledge useful for developing applications with 3D rendered elements.

Lastly, Android development is a continued interest, having rooted and installed

custom firmware (ROMs) on many devices, and getting familiar with the OS and the main Android-PC interaction tool – ADB – developing applications is the next logical step – and one which comes with enthusiasm. Experience with Java meant the barrier of entry to developing on this new platform was lower than with other alternative technologies.

## 1.4 Relevant Projects

As preparation before embarking on this project, a few other systems were looked into for inspiration and getting a feel of the market this project would be entering.

OpenAero [1] is an extensive tool for designing aerobatic sequences built by Ringo Mass. It takes the form of an open-source project web application. Among many more complex features, it primarily provides OLAN interpretation and translation of OLAN into Aresti, with diagrams and catalogue references of manoeuvres. It is not so much a tool for visualising acrobatic flight as a tool for people who already understand the manoeuvres and want to design flights combining them, but it offers some useful insights into the shapes of manoeuvres and the OLAN system.

The OLAN tool [2] is a Windows application developed by Michael Golan and is the point of origin for the OLAN system. It provides a functionality similar to OpenAero, only in a less modern context of a dedicated application, but it does have a catalogue interface, allowing users to browse through available manoeuvres.

## 1.5 Platform and Technology

This project is aimed at the Android platform. The decision was made based around several main factors – convenience of hardware platforms available, market share, ease of development and support for the 3D graphics this project uses.

The final result of this project needs to be conveniently accessible. It's highly likely that the user will want to use the application where they practise the flying – “in the field”, so to speak – so the project needs to be available on a portable medium. Now, computers can be considered portable and accessible, due to laptops, however, this isn't the most convenient medium, as they're often large and cannot be easily used without something to put them on. A mobile device – such as a phone or a tablet strikes as an obvious remedy. Another relevant, interesting statistic which supports this platform choice is that usage of mobile devices is increasing, while that of PCs is decreasing [3].

Choosing between the various mobile operating systems was not a difficult decision. Android has a larger user base – 76.6% of the mobile devices shipped in the fourth quarter

of 2014 were Android devices (this has been a continual trend for a while) [4] – when compared to the iOS, Windows Phone, and Blackberry platforms. Access to devices (of different formats) for testing, generally familiarity and openness development procedures were also considerations which tipped in the favour of Android.

The other side of the background research was looking into the Android platform and development. The most significant part of this, on the development side, is the implementation language. It is possible to develop on Android using a wide range of different languages, with official toolkits allowing development in C/C++ via the Android NDK, and third-party libraries supporting Python, Ruby, Pascal, Lua and more. The most commonly used language for development on Android is Java, as the official SDK is Java-based and this reduces the need for relying on third-party libraries, and should make development easier, with access to a wider range of support and the language being familiar.

It is also crucial to ascertain that the platform could support the requirements of the project – rendering 3D graphics. Android supports OpenGL ES, with the most common version available being 2.0. The platform has libraries providing higher-level interfaces for rendering 3D graphics and building scenes. A few of these were looked into, though no real incentives were found for what would be a fairly simple scene in terms of both shaders (lighting etc.) and content. The decision to go with OpenGL ES was based on experience with WebGL, which is likened to be very similar to OpenGL ES 2.0.

A possible issue with Android is its history of being a notoriously fragmented platform, particularly in terms of OS version – a factor which trickles down to the versions of libraries as well. It is then important to decide which version of Android – and thus which Android SDK level – to aim the project at, as this will impact the design through the availability of various APIs.

In terms of hardware, the Android platform posed some different challenges to those encountered when developing for a PC – the hardware gamut of mobile devices is quite different to general computers. This difference mainly hinges on greater variation in screen sizes (both physical and resolution) due to different form factors (phones/phablets/tablets). Also to be thought about is the fact that the general power of devices is across the board lower than comparison to PCs, so that needed to be taken into account. Both the UI and the complexity of the visualisation should take these factors into account. With Android, these performance issues could be somewhat mitigated by restricting access to the application by giving it a minimum API level. Newer devices tend to more power and newer API levels, so an application can sort of be targeted in this way.

## 2 Analysis

The project, at a rudimentary level, can be broken down into three main components:

- The internal representation of the manoeuvres.
- The translation of flight descriptions into that model.
- The visual modelling of that flight.

While these three components cover different aspects of the system, it should be noted that they don't necessarily have to be distinct in the actual design and implementation. A prime candidate for this is a high level of cohesion between the representation of manoeuvres and their visualisation component.

### 2.1 Representation of Manoeuvres

The first task highlighted by the research was that of developing a representation of the manoeuvres in-application. There are, of course, a number of different ways of doing this, including hard coding every manoeuvre – but that is not a very effective way of doing things.

The most logical approach is to analyse each manoeuvre and find a set of the highest common factors of movement between them all. As the standard catalogue consists of a lot of different manoeuvres, this is an extensive task, but there are definitely some common features between various ones, for example: components such as a short straight line forwards and a turn upwards by a few degrees are used in loop-the-loops, diagonal movements upwards and many others. In this process, some parameters, such as amount of change in direction need to be derived. For example, with a manoeuvre set consisting of a straight up vertical movement, and one at a diagonal climb, a component's upward turn angle should not be ninety degrees. It's logical to clamp this value to a discrete range to keep the components simple. By changing the length, different sharpnesses of turns can be recreated.

Alongside this, another perspective which can be taken is the interpretation of manoeuvres in the context of actually being flown by an aeroplane. This is a useful content because the manoeuvres are grounded in reality, and with it in mind it is obvious that manoeuvres can only exist if they can actually be performed. Thus the movements in a

manoeuvre are limited by the mechanics of flight. From this, we can distil that the components of the manoeuvre should only represent ways a plane can move – a plane cannot suddenly switch from flying straight to flying upside-down – so a series of smaller movements can be built up, using operations such as a realistic change in pitch, yaw, roll and movement forwards (or perhaps backwards, in some cases). A component can then be thought of as a bit like the plane's velocity at a point – with a direction and a length.

Further on this, each piece of the aircraft's movement and progression through a manoeuvre is reliant on its previous position and orientation. Any set of simple components, representing an aircraft's movement, needs to be a cumulative progression.

However, this cannot be completely accurate, as the ability for an aircraft to influence its position and orientation while flying straight line forwards is different to doing the same thing but upside down. The influence of both gravity and the thrust generated by the wing are two large factors which need to be considered – or ignored. Other factors worth thinking about are how movement in each axis is different – generally, aircraft can adjust their yaw to less effect than its pitch and roll – and how the characteristics various aircraft differ, such as how much influence its ailerons exert on its direction – how quickly it can turn etc. These issues are complicated, and require a complex model to be able to factor them in properly. Since the aim of this project is not to build a realistic flight simulation – of which the issues noted barely scratch the surface – but to simply visualise the flight in a comprehensible way, these problems can be largely ignored. It is a reasonable assumption that at the scale offered by this project, these factors would have very little influence anyway.

A more pressing restriction of this model is that it only allows the aircraft to move on one axis – forwards/backwards of the direction the plane faces. It is entirely possible that some manoeuvres include momentum-based movement, deliberate stalls and other non-standard ways of moving an aircraft. This feature of movement could also present a challenge to the visualisation component, and not taking it into consideration may limit the manoeuvre set of the project. However, it is difficult to develop a model which allows for this sort of motion. Breaking away from the model described earlier to support this sort of thing can bring in issues such as allowing completely erratic movement of aircraft, with no logical relationship between points in its movement.

It is an important to recognise that the manoeuvres in the catalogue are data, and thus their representation should not be hard-coded into the application, and instead stored separately, as a resource data file. Making this catalogue file recognisable – using a fairly standard data format such as XML – allows for simple extensibility and allows for building on the catalogue and adding extra manoeuvres. Encoding the manoeuvres in this resource

file scheme is likely to be a fairly manual job (difficult to automate), and with a large catalogue of manoeuvres, so it could take a while. This catalogue file is a useful resource though, and it is reminiscent of the Aresti catalogue/directory the OLAN (application) provides. Making this available to users as a resource for building flights is a sensible feature idea who don't know the kinds of manoeuvres available for aircraft fly. It also has programmatical use in the application, as a entity which can be queried with a manoeuvre's name to get the representation of an manoeuvre.

## 2.2 Flight Description Interpretation

The interpretation of the flight description relies first on deciding which flight encoding method (Figure 1) to use – Aresti or OLAN. Creating a new system should not be entirely ruled out, though really there's little point in reinventing the wheel. After a bit of research, including investigating views on a model aircraft flying site [5], some ideas of the advantages/disadvantages and suitability for this project of each system were developed and tabled in Figure 2. Weighing these up is important for choosing the most appropriate one.

Attribute	OLAN	Aresti
Brevity	Letters with various modifiers	Diagrams or numbered catalogue references
Familiarity to users	Fairly new system	Standardised by the FAI
Openness	An open system	Catalogue must be bought
Customisation	Modifiers allow for changing manoeuvres hugely	Set of manoeuvres limited to the official catalogue

*Figure 2: Weighing up the flight description systems*

Despite the advantages Aresti offers in terms of general familiarity and standardisation, the positive aspects of the OLAN system are more relevant to this technical project, thus the decision was made to go with that system. The ease of user input – simple strings of characters, instead of diagrams/catalogue references – and open availability were strong factors in this decision. A compromise can however be struck, by providing the Aresti catalogue references alongside the OLAN figures (an incomplete set can be found via OpenAero) in the manoeuvre catalogue, therefore allowing users to find manoeuvres by Aresti as well as OLAN and allowing users to build connections between the two systems.

Parsing this string into the program's internal representation of a flight is fairly simple

task, despite the fact that there are quite a few different modifiers available in the system. Figure 1 shows some flights expressed with OLAN, and Figure 3 explains the system in action.

An example flight – “6% ++ `` `k+ 1 v ``” – denotes a flight of three manoeuvres:

- A “k” manoeuvre (shark tooth, vertical up and a diagonal down) scaled by six, which has an extra two steps to its intro and one to its behind and the first group (vertical up line) scaled by a quarter.
- A simple full roll.
- A with the second group (vertical up line) scaled by a third.

*Figure 3: An explanation of an OLAN flight description*

An added complexity of using OLAN is the modifiers which the system supports. They can change the way manoeuvres in quite a dramatic way that cannot be expressed in Aresti (which is more limited by its catalogue). Any representation of a manoeuvre needs support for these modifiers. Some of these are fairly trivial – full feature scaling and entry/exit scaling – but on the other hand, some are fairly complex. The group scaling is a bit tricky, with certain parts of the manoeuvre being scaled. These groups vary between manoeuvre, and really needs to be tied into their representation. Also, these modifiers need to be interpreted in the flight description as well.

- `(\\d)% ([\\+,-]*) ([` ]*)(\\w*)([` ]*)([\\+,-]*)`
- `{number, full scaling}% {pluses, entry length}{backticks, inverse first group scaling}{OLAN figure}{backticks, inverse second group scaling}{pluses, exit length}`

*Figure 4: Regular expression for extracting parts of an OLAN figure*

Most of the OLAN figures can be quite easily derived by splitting the statement by spaces to get each figure, and then using regular expression groups to extract the various components of each manoeuvre (see Figure 4), should the modifiers exist in the string. However, there exists the case of full manoeuvre scaling, which is separate from the figure, so this needs to be handled slightly differently with going to the next figure for and scaling it if it's relevant.

## 2.3 Visual Modelling of a Flight

The visualisation aspect of the project is arguably the most complex part, as it's not really constrained by any hard and fast rules, like previous parts of the application, and it allows room for creativity, with a wide range of different possibilities. Most importantly, the

way the visualisation expresses the flight should be easy to understand and offer value for its performance demands (there's little point in having something pretty which is unclear to the user).

Watching an aircraft flying an aerobatic procedure, as stated before, does of course offer some value in terms of educating viewers on how to do the manoeuvre. This can be replicated in the application as a method of visualising the flight, but can also be improved upon. In the real world, there isn't many more options, other than a technique occasionally used by stunt pilots – smoke trails. These leave a (comparatively) lasting impression of the tracks of the plane and, while they're not used to bring about an understanding of the manoeuvres, they can be an inspiration for developing a visual solution.

A world without rules and controls, without borders or boundaries. A world where anything is possible. Where we go from there is a choice I leave to you.

*Figure 5: Thoughts on a blank canvas [19]*

The options for building a 3D scene and its rendering are almost limitless in the blank canvas which a graphics context is (Figure 5). There is no real reason why the visualisation should go for a realistic look, thus a more abstract, informative approach can be taken. Using the level of the aircraft's wing as a (surface) plane, a full flight can be thought of as a ribbon, with the area behind the aircraft that it has been through filled in, similar to the smoke trails, but not necessarily temporally bounded by inevitable fading out. There is an issue that this ribbon may be too abstract, and with no reference points in a blank 3D space, it can be difficult to understand. Adding a reference (surface) plane which acts as the ground would be a good way to tackle this issue.

A ribbon in space like this does not necessarily map very well onto the actual movement of an aircraft. There exists problems such as the direction not being clear, and with complex flights it might be difficult how the ribbon is all linked. Animation is a tool which can be used to combat these issues. This is a logical idea, considering that it harks back to the subject matter – (most) planes are continually in a state of motion during flight. Showing the progression of the flight from the beginning to the end is a good way of avoiding it becoming a bit too abstract. There are, of course different ways of doing this too, but following on from the smoke trail ribbon idea explored above is a sensible method. Having a full flight drawn behind the aircraft as it progresses through the flight is an option – smoke ribbon lasting forever – and so is having a flying wing look – smoke ribbon only lasting so long.

As a 3D rendering is derived from a 3D environment, it would also be beneficial to



offer some control to the user to be able to manipulate the perspective. This can come in the form of rotation, translation and zooming. This would vastly improve the visualisation of the flight, giving it a truly 3D feel and allowing the user to build a better comprehension of the manoeuvres.

Changing the position and orientation of the view will likely require a user input. Due to the platform of choice being Android, the application will be running almost exclusively on touch screen devices. This constraints the inputs available, and needs to be factored in when developing these ideas.

## 2.4 User Experience Focused Features

Besides the basic core of the project – the three features detailed earlier – things which make the application more useful, accessible and usable are important for ensuring the application is actually valuable. The scope of this particular area can be quite wide, with a near endless list of features which can listed under usability and accessibility. It's important to keep the feature list focused.

The vast majority of accessibility and usability based features make up part of the user interface in general. This UI should be logical and intuitive, so users are never confused and can predict with accuracy what each UI element will do when used. Another way of improving this accessibility is to provide a series a set of documentation – help text – accessible in-app, which describes how to use it. This is an unobtrusive way of explaining features to new users, but allowing more experienced users to work uninhibited.

In terms of extra features for making the application more useful, having a store of previously written flights benefits the user greatly, preventing them having to rewrite flights every time they open the application. Being able to edit them by tweaking the OLAN encourages the development of manoeuvre sequences.

There are various parts of the application which have an element of user preference, with no clear, right answer. In these cases, it is possible that the user might having a choice of a few different options useful – things like whether or not to correct potentially invalid OLAN, a speed for the animation etc. Providing a settings menu for users to select their preferences is a good solution for this.

# 3 Process

## 3.1 Introduction

The process chosen for the development of this project takes into account attributes of both the project's technology and its developer. The resultant methodology does not confirm to any standard, strictly defined approaches to software development and instead can be considered a bit of a hybrid methodology, taking features and values from various different ones. The decision to not rely on any fixed, formal methodology is thanks to their tendency to be weighted towards team development (this project has a single developer), and being unable to find a regime which development could confidently stick to.

The process used can be categorised as an iterative, agile-based methodology. This seemed most appropriate for working with unfamiliar technology, as it is difficult to estimate accurately how long jobs will take with no prior experience in the field, and this is often crucial for building up a plan. Agile ways of doing things provide a flexibility useful for adapting to any unexpected issues ran into.

The main methodology which influenced the process is the Feature Driven Development model [6]. This starts with analysing the problem, creating a feature list, building a bit of a plan, and then working through that feature list in an iterative manner. In this process, most of these components were used, like the feature list and iteration.

Analysis of the project brief and discussion with the supervisor were useful for building a feature list. Though not all finally implemented features were apparent at that early stage, the most important aspects of the system were obvious, the agility and repetition of the methodology supported the dynamism of this list. It was important that this list be ordered based on the focus of the brief – the visualisation – and the UX features having a lower priority.

FDD also suggests a brief period of overall planning in its primary phase, to build up a rough system architecture which incorporates the full feature list. Inexperience with the Android system, meant that this kind of architecture plan was difficult to assemble, even with the research into good Android development practices. In place of this, the design of the project was implemented in a evolutionary manner, with a high value placed on simplicity. Ensuring this meant recognising the requirement of (and enforcing the practice of) merciless refactoring.

## 3.2 Key components

Progress tracking is an important part of any development procedure. Keeping accounts of work done on a regular basis is a simple way of doing this. Throughout the project, daily notes were kept in a notebook, with a log of work carried out that day and the workings of various algorithms/designs. These notes were consolidated into weekly development diaries to provide a summary and an easier way of monitoring progression. Alongside this, a weekly demo of the application is possible, utilising screen recording software, offering a clear insight into the progress between weeks.

Testing was of course part of the development process. This can be quite tricky due to the flux in the design, the graphical nature of parts of the project and the interfaces available for testing with Android. The testing procedure is covered in more detail later, but, as an overview, tests were written for a feature on its completion, once the interface for the feature had been finalised, and were used to ensure that subsequent refactors and feature addition did not break them.

Using a version control system is another important part of the process. It's crucial for maintaining a backup of the changes made in the code so any mistakes can be recovered from. A general scheme of version control usage during development is to commit every time an important piece of design is changed, such as when a refactor/feature is complete, and never to commit a broken build. Using a version control system also allows for automating backups which increases the security of the codebase. Git was used for the version control due to familiarity and the simplicity of using free resources like GitHub [7] and Bitbucket for remote storage.

## 3.3 Preparation

Before any development work could begin, there was a number of different things which needed to be prepared.

Taking ideas from the FDD software development methodology, the research and analysis inspired a rough feature list which guided the iterative development of the project. This is largely built from the analysis covered above, with work stemming from the model outwards.

Key also to starting the project was getting to know how to develop for Android. The very rough prototypes at the beginning of the project bore no resemblance to the final product, but were instrumental in building understanding of APIs – how to construct activities and how to use OpenGL ES's Java bindings. Google offers some tutorials and a

comprehensive set of documentation [8] which were used for initial learning period, as well as for reference throughout the project. While Google offers tutorials on OpenGL ES, theirs aren't the best, so other sources were looked at [9].

### 3.3.1 Development Environment

One of the key prerequisites for any software development is an environment to work in. Many technologies, including Android, have requirements which are not part of a standard (computer) system, and thus these needed to be setup.

The two main prerequisites of Android development are the Java and Android SDKs, each of these comprising of a set of development tools for building applications in their respective technology. These need to be downloaded, installed and configured.

Application code can be written solely on top of these, using a text editor and direct interaction with the command line interfaces each of these provide. However, this can be an arduous task, and certainly a steep learning curve for a developer inexperienced with the platform. Instead, it is logical to look into what IDEs are available for the platform, and what level of functionality they offer to improve code/speed up development.

There of course exists a range of options available for IDEs, with dedicated ones and plugin to add support to others. Out of Eclipse, the Android-based AIDE, IntelliJ IDEA and Android Studio [10], Android Studio was chosen due to its designation (by Google) as the official Android IDE. Each of the different choices offers more or less the same feature set, but the decision to go with the official, first-party one was influenced by it providing the simplest introduction to Android development with the Gradle build tool and Git integrated. Gradle is a really useful tool which automates many of the tasks necessary to build an application successfully, and thus greatly speeds up development time.

## 3.4 Iteration

Development proceeded with an iteration time based loosely on a week – the time between the development diary posts. At the beginning of the development week (Wednesday), a feature was decided to be worked on – usually the obvious next feature came up in the development diary post – from the list of features developed, in a general order of importance.

Work progresses on this feature though the week, with efforts focused on developing a working prototype first, and then then considering its place in the design, and refactoring to improve the design. This last stage involved writing code documentation (in the Javadoc style) and writing some tests for the code.

During the week, a more frequent daily book was kept, with a focus on working out problems, writing general notes and planning the work of the next day. This was useful to focus efforts and work out mathematical/design problems.

## 3.5 Supervision

While it was initially proposed that there would be weekly meetings with the client/supervisor to discuss progress that week and clarify the next round of features, this was deemed unnecessary after a few weeks. Thanks to the level of understanding of features demonstrated through the analysis, the interpretation of the task shown through the feature list and the implementation of work already carried out in the prototype stages, the client and developer were satisfied and confident that less frequent meetings were sufficient.

## 3.6 Summary

In summary, the primary values of the methodology used:

- Iterative model, with the most important features put first.
- Flexibility in design throughout the project (forgiving platform inexperience).
- Simple design through merciless refactoring.
- Continuous integration testing throughout development.
- Documented code throughout development.
- Progress tracking through weekly summaries.
- Intelligent use of version control.

# 4 Design

## 4.1 Introduction

Due to the process by which this project was developed, the design and architecture are inherently the products of an evolutionary process. This means that at any one point in the development of the project, the overall design was subject to change. Even at this final stage of the project, there is still an element of flux with the design going forward with any further extensions. However, that said, the pace of large-scale changes to the overall design slows down during development, with the arrival at a (somewhat) simple, yet extensible design. The design described is the end product of many of these different designs.

It is worth bearing in mind that there is seldom a perfect design – there are always alternatives, but at times none of them are objectively better. Care is taken to make sure this is considered during any redesigns and refactors so that they are actually worth doing, and arriving at this design is the conclusion of lots of thought and consideration.

### 4.1.1 Inspiration and Influence

The designs used across the project at various levels of architecture were inspired by well-established design patterns [11] and frequently used Android development practices [12]. The latter was particularly important in its research and implementation, considering the lack of any prior experience with the platform.

During the background research and problem analysis, some ideas came with clear suggestions to the design of its respective component – for example, the manoeuvre's breakdown into a list of components. This research also uncovered some possible use cases for design patterns and various other solutions. While it was an evolutionary design, these apparent components of the architecture were present right from the beginning of development (namely, the model). It should be noted that their presence throughout the development was not due to being shielded from any of the refactoring, rather that they were simply part of a logical, effective design.

Naturally, another influence to the design was the Android SDK. This has various restrictions on things, such as file resource access, and different ways of doing things compared to standard Java. As mentioned before, by utilising common software development knowledge and good practices these design issues could be understood and tackled to arrive at good working code.

The design is also based around the fundamental concepts of the paradigm of the implementation language – Java, an object-orientated language. Programming in this paradigm prizes – amongst other concepts – abstraction, encapsulation and polymorphism, and these features were made use of to great extent throughout the design.

For the visualisation aspect of the project, various different libraries were investigated. Using these would exert an influence on the design, but also a lack of third-party libraries and a direct implementation through OpenGL ES impacts the design too. The effects of this low level interaction are somewhat mitigated through the abstraction (and polymorphism) offered by object-orientation.

### 4.1.2 Evolution

As discussed earlier, the development taking a feature-driven agile approach meant that features were worked on more or less sequentially, so earlier designs were less feature-complete than later ones. With a focus on simplicity, earlier designs reflected that, particularly in the prototype stages, where the view and controller sections of the design were a lot smaller and there was a complete lack of managerial classes.

Development started with prototypes of an OpenGL visualisation, getting the hang of how to build an OpenGL program in Android and draw vertices, then moving onto restructuring classes to provide abstraction. Then the model was built to store a representation of the flight, then an XML expression of this and the manoeuvre catalogue parser. From from this basis the rest of the application was developed.

## 4.2 Overall Architecture

Simply put, the overall architecture for this project is inspired by the Model View Controller (MVC) design pattern, with the majority of classes fitting into these three categories (and aptly named packages) as Figure 6 shows.

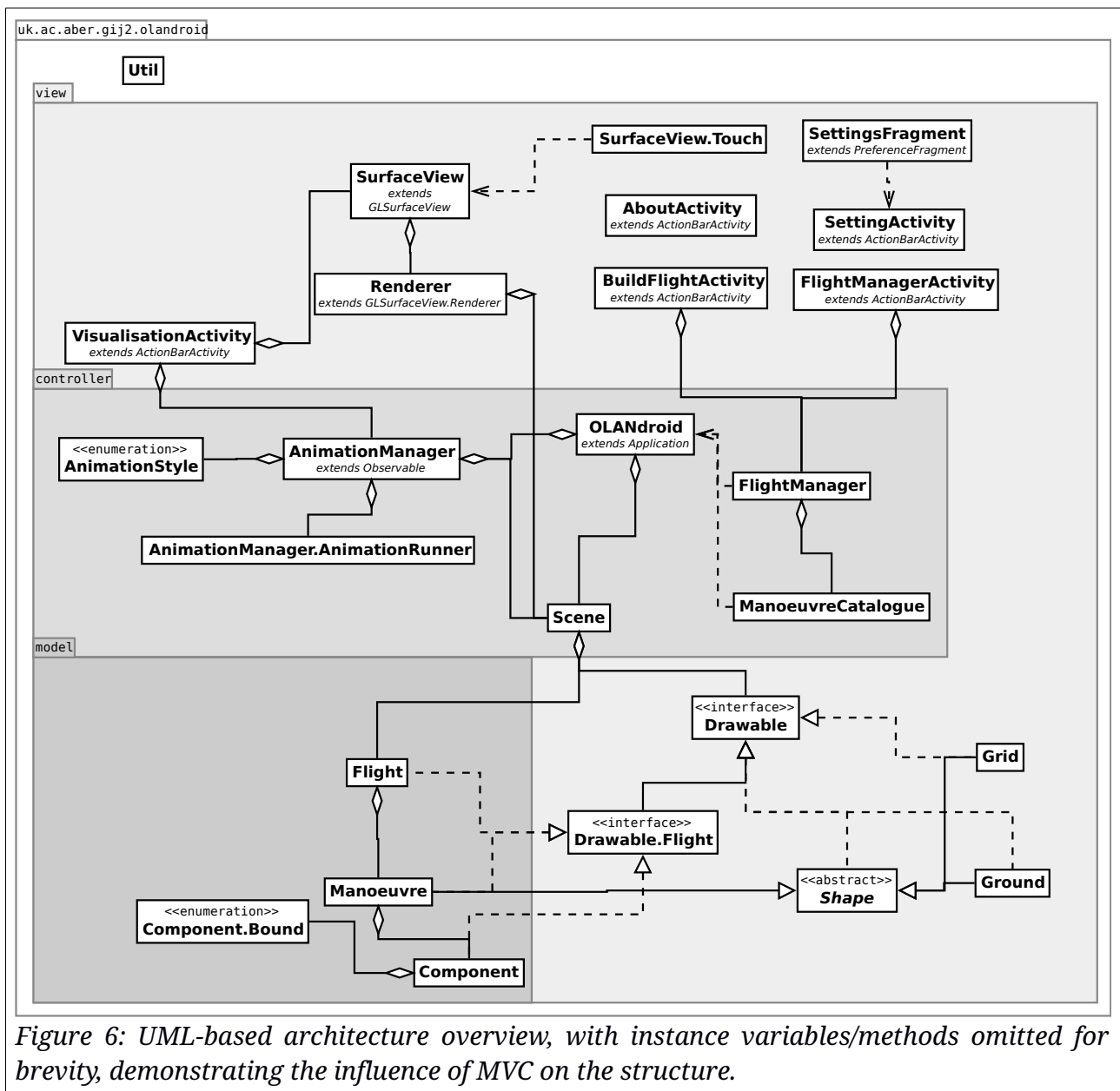


Figure 6: UML-based architecture overview, with instance variables/methods omitted for brevity, demonstrating the influence of MVC on the structure.

As Java is a strictly object-orientated language, Figure 6 also serves to offer an overview of the classes involved in the architecture. The diagram also notes some nested classes/enums/interfaces. Such are nested as such as a mechanism for organising and, in some cases, limiting their scope.

## 4.3 Design in Detail

### 4.3.1 Model

The model part of the design mainly encapsulates the representation of the flights, which includes their constituent manoeuvres etc. The model also maintains the state of the flight, storing any modification to them, and responds to changes marshalled by the



controller. It's considered a pretty passive model in this application, with it not having any outward ties to the controller or view, and so changes to it are observed and acted on on changes triggered by the controller..

### a) Pure

The “Component” class, in its most basic form, is the coded manifestation of the highest common factor of movement in manoeuvres, also based on the principal movement of an aircraft on its axes. Its most important features are the values for pitch, yaw, roll and length, which are used for building the drawable shape of the component in vertices and constructing the matrix which specifies the transformation from the beginning to the end of the component (this matches the drawable vertices).

A level up from that, the “Manoeuvre” class represents a flight manoeuvre through an array of “Component”s. Its default is an unmodified manoeuvre, with methods available for applying the various OLAN modifiers, like scaling its size (of the full manoeuvre, or specific components) and extending the entrance/exit. It stores an OLAN figure (inclusive of any modifiers) and a name which describes the it.

The “Flight” class is reminiscent of the “Manoeuvre” class, with it containing a list of “Manoeuvres” much in the same way that that class has a list of “Component”s. Many of its features are similar to the movement, with it also keeping a name for the flight when it saves.

### b) Compromise

This model part of the design is perhaps muddled a little by the fact that it also incorporates elements of the view – visualisation – aspect of the design. This comes though most significantly in the presence of the “Shape” abstract class which the “Manoeuvre” extends, and “Drawable.FlightPiece” interface which all of the model adhere to. This is a consequence of the high value placed on the simplicity of the design since the prototyping phase and sufficient abstraction being developed around the “Shape” which provides the drawing facilities.

For the model, the “Drawable.FlightPiece” interface is a key part of this design. It inherits from the “Drawable” interface (which simply specifies draw method taking an initial matrix), detailing the extra functionality required of the pieces of a flight. The methods which it requires – an operation for getting a transformation matrix from the beginning to the end etc. – are designed to be reusable, and valuable to the lower level implementation of the flight as well (“Manoeuvre” and “Component”). This is particularly noticeable when comparing the design of the “Flight” and “Manoeuvre” classes, as both

contain lists of constituent parts (manoeuvres and components respectively) and is useful for passing method calls down through the levels of the design.

The “Shape” abstract class, which provides the functionality for drawing things in the visualisation and goes hand in hand with the “Drawable” interface, is extended by the “Manoeuvre” class. The “Manoeuvre” class uses the vertices and matrices of its components to build up a list of vertices of its own, which are then used in the visualisation in methods defined in its “Shape” superclass.

In previous designs, the “Component” instead extended the “Shape” class, and drew themselves, but this was a lot slower, as a lot of the matrix transforms of vertices had to be redone every time. Combining these into the “Manoeuvre” means that this vertex/matrix multiplications can be cached and efficiencies made.

### 4.3.2 Controller

The job of the controller part of the application is to mediate changes from the view to the model. In this design, the task is performed by several manager classes which each deal with different aspects of the feature set.

The centre of the controller is the “OLANdroid” class. This extends the “Application” class, which is described as a “base class for those who need to maintain global application state” in the official Android API. This hints at its use in this design, with it acting as a supervisor, passing state changes through and down to the model. It also has the advantage of being in the chain of inheritance belonging to “Context”. Contexts are important in Android development, as they provide an interface for accessing application-specific resources and functionality such as moving through the UI. The decision to subtype the “Application” class was made mainly for this reason, despite there existing reasoning for the alternative, singleton-based approach [13], it was important that there was access to a “Context” for accessing resources such as the manoeuvres data file, the saved flights file and textures. “OLANdroid” ends up being a singleton due to the nature of the “Application” class in Android anyway.

This singleton idea was carried across to the majority of the other sibling controller classes, with them also being singletons. This design pattern is used to restrict the instantiation of each to only one object, so the point of responsibility for the model classes are always easily referable, making maintaining the state much easier. An earlier design included all of the functionality of the controller classes in the “OLANdroid” class, but this was eliminated after the identification of the 'god-object' anti-pattern where one class had way too much responsibility. The “OLANdroid” class still maintains the responsibility for initialising the controller classes, as it is the first class to be instantiated on starting the

application.

Another duty the “OLANdroid” class has is to listen to changes in the preferences and apply the relevant changes. Its position in the architecture of the system, presiding over the model, makes it ideal for this job, so any preference changes can be simply applied. Preferences are implemented in the standard Android way, utilising XML to detail the various preferences and what options are available for each.

### a) Managing Flights

The “FlightManager” class deals with the organisation of flights. It provides the OLAN interpretation, height correction and flight building functionality, a store of flights loaded as well as adding new ones, deleting old ones, and the loading and saving of flights to a data file.

The interpretation of the OLAN relies on the “ManoeuvreCatalogue” for checking the validity of various figures found in the OLAN strings, so it has reference to that catalogue. The “FlightManager” also performs the flight correction, with adding correcting figures and rebuilding the flight with the addition of a correcting manoeuvre (manoeuvres can be defined in the catalogue to state if they are eligible for use as a correction manoeuvre on whatever axis) until the flight doesn't dip below the horizon. This is an optional feature toggled using the preferences menu.

The “FlightManager” gets access to the data file via the initialisation in the “OLANdroid” class, then it loads flights from a simply formatted file which stores the OLAN for a flight and a name. It creates these flights and stores them in a list, which is used by the UI to populate the flights list on the first application screen loaded.

### b) Manoeuvre Catalogue

Similar to the Aresti reference catalogue, the “ManoeuvreCatalogue” provides a list of manoeuvres, indexed by OLAN figure. To build up this catalogue, it loads in and parses the manoeuvres data file (Figure 7) detailing a range of manoeuvres. Manoeuvres are stored in a “LinkedHashMap”, and a “Manoeuvre” can be retrieved by querying the catalogue with an OLAN figure as a string. This data structure is chosen for a number of reason:

- Flexible in size – vital when parsing the unknown size of the catalogue file.
- Associative – useful for a catalogue where things need to be looked up with keys.
- Predictable iteration order - great for maintaining consistency in the catalogue when it is adapted for the user interface.

As the manoeuvres in the data file are organised by category, alongside returning all of the manoeuvres, the catalogue also supports returning a subset of manoeuvres based on

the passed category. This is useful for organisation in the user interface.

### c) Animation Managing

The “AnimationManager” class handles the animating of the visualisation. It has functions for playing/pausing the animation and setting the current progress (between zero and one), which is then passed down through to the flight etc. This class makes use of the observer/observable design pattern, with it extending the “Observable” class and triggering events when the animation progression changes. This allows listeners, like the user interface, to respond to the progression of the animation.

For running through the animation progression, this class makes use of a private inner class which implements the “Runnable” interface, this is used to thread the animation progression and maintain the responsiveness of the rest of the application while the animation is playing. The animation is such that it runs at a constant speed, regardless of the length of the flight. This speed can be changed with various options available in the preferences. It should be noted that animation steps do not trigger draw frames in the “SurfaceView” – they're not synchronised. They're kept separate for a few reasons detailed later, one of which is to maintain the distinction between the view and the rest of the design.

### 4.3.3 View

The view part of the application generally covers the user interface and the tools needed for the drawing of the visualisation. As mentioned, the “Shape” and “Drawable” components do have ties to this section, but the distinction here is that they comprise of content to be drawn, rather than the tools for drawing.

One of the main themes for this part of the architecture was the avoidance of any functional feature code, but instead to provide capability for drawing the visualisation, give the user with the output they want, and pass the user input to the controller part of the application.

### a) Activities

Starting of with the main UI component, all of the “Activity” classes in the view extend “ActionBarActivity”. This class is a subtype of “Activity”, which is basically a page of the user interface – a screen's worth of UI elements. Views are these elements in the page's layout.

The layouts of all of the activities can be built programmatically, or are expressed in XML, parsed and assembled by the activity. While layouts can be totally built and styled in the initialisation of an activity using code, it is much clearer and concise to use the XML.

Simple GUI tools are available for assembling these too.

The decision to use the “ActionBarActivity” as opposed to a completely blank one was led by the need to have constant UI elements easily available – a back button, activity name and some menu buttons (help and settings). Some activity-dependent menu buttons are also on the menu bar, such as the new flight button in the “FlightManagerActivity”.

The first screen you are greeted with on opening the application is the “FlightManagerActivity”. Besides the action bar, the only content in this activity is the list of previously saved flights. The “FlightManager” is queried for application's loaded flights (loaded from the data file on initialisation) and then this list is adapted into a list in the UI, with various operations available to users on tapping.

Building/editing a flight is done through the aptly named “BuildFlightActivity” class, which is activated by starting a new flight, or editing one (which sets the OLAN string to the current flight's OLAN). This contains a list of manoeuvres sorted up by categories, which is adapted from the “ManoeuvreCatalogue” (as one list of flights would be very long); a dropdown box of various manoeuvre calendar, changing which repopulates the manoeuvre list with relevant ones; a text editing box, allowing for the direct input of OLAN; and buttons for building the flight and directly adding some modifier characters. The list of available manoeuvres listens for presses, on a short press it adds the OLAN for the manoeuvre in question to OLAN text entry; and on a long press it brings up a dialog box displaying a relevant Aresti diagram. The class uses the “OLANdroid” class to pass the OLAN string entered across the “FlightManager” to interpret the flight.

The “VisualisationActivity” is pretty empty except for the custom “SurfaceView” element, and a seek bar which overlays it at the bottom. This provides some input and user feedback for the animation progress, with this class listening to the “AnimationManager” for animation progression and updating its UI in response to updates. The action bar has elements for the operations of playing/pausing the animation, saving (with a name, brings up a dialogue box), and editing the current flight – which takes the user back to the “BuildFlightActivity”.

## **b) Surface View**

The presence of the “SurfaceView” class, as an extension of “GLSurfaceView” instead of simply using that, is fuelled by the need to have a surface to listen to touches with. The “SurfaceView” class provides this, as well as being a canvas on which to draw the scene. Touches are interpreted into one of two different control schemes available and passes to its “Renderer” to manipulate the 3D render.

The control schemes are set via the preferences menu, and implemented using an inner class to represent a touch. This small inner class provides functions for euclidean, gradient and finding a vector between two, making the control scheme code much easier to understand.

### c) Renderer

The “Renderer” is the central tool which renders the visualisation, extending Opening’s “GLSurfaceView.Renderer” and providing the same functionality and more. It is responsible for compiling the shaders (GLSL) and building an OpenGL application, loading textures, building the view matrix and drawing the scene.

The “Renderer” also handles the view movement/rotation, making use of Android's OpenGL libraries for the task of generating a new view matrix from the multiplication of these view movement matrices.

The “Renderer” class also manages the loading of textures resources. It creates an accessible array of texture ID handles they're all once loaded, making using textures simple to use by passing one of these reference IDs to the shader, done by the “Shape class”.

The renderer is set to continuously re-render the scene, rather than tying it to a specific cycle, or manually triggering draw calls. The reason behind this is that the renderer can be a bit more independent from other events (animation, view manipulation). Tying a rendering cycle to these could end up with placing a lot of draw calls in very rapid succession, overloading the system and damaging performance. Instead, continuously re-rendering the scene allows the application to go at its own pace, within the limits of the hardware, and still reflect the state of the scene at the point of drawing.

### d) Shaders

Both the vertex and fragment shaders used in this application are very simple, only providing functionality for using basic vertex colours or, as an option set using a uniform boolean, texture sampling using whatever texture ID is set. This is due to the stylistic choices made for the visualisation, as well as for keeping the visualisation efficient and not ruling out low-powered devices.

The shaders are stored separately in GLSL files as “raw resources” in Android's resource management system. The “Renderer” loads these files using methods in the “Util” class, then compiles them as shader programs for use for rendering the scene.

## e) Shape

The “Shape” abstract class encapsulates the interaction with rendering part of the program – OpenGL and the shaders – so any extensions to it do not need to manage this themselves. In place of that direct interaction, it provides an interface which supports the drawing of planes (or empty polygons) through a list of vertices and a draw order, with a few extra optional features such as front/back surface colours and textures. It uses these vertices to build buffers which are passed to the shaders which iterate over the objects to draw and render them to the “SurfaceView” via the perspective defined in the “Renderer”.

Due to this, the “Shape” relies on the “Render” to have an compiled and initialised OpenGL program (which thus relies on the shaders), which is accessed through a public reference. Since the “Renderer” is initialised by OLANdroid at application start-up, this OpenGL program reference can be relied upon (no draw calls are made until the “SurfaceView” is initialised, way after start-up). To further reinforce this, the draw method in the “Shape” class will be skipped if the “Renderer” is not yet ready, preventing any potential mishaps.

An alternative design for building the drawable shapes was to have a class separate from the “Component” and “Manoeuvre” which handled the drawing was considered. This would come as either a sort of parallel class to the drawable bits, or a managerial class which interpreted the drawable pieces and drew a shape. These ideas were abandoned due to the increased complexity in the design and increased work of assembling vertices rather than using a reference catalogue of them, as well as the larger memory footprint.

## 4.3.4 Resources

### a) Manoeuvres

The manoeuvres are encoded in XML. The reasons for this file format being selected are its ubiquity, strength of Java XML parsing libraries, consistency with the rest of the Android API (it's used for layouts, string lookup tables, theme definitions, preference declaration and plenty more) and human readability. Also bearing in mind the fact that the set of manoeuvres encoded is not a complete set (due to the lack of access to an Aresti catalogue), extensibility was also something which was considered during the creation of this file. As the representation of manoeuvres is quite simple to understand, it would not be beyond most people to be able to contribute to the catalogue, or even define their own custom manoeuvres.

```

<category name="Single line element figures">
  <manoeuvre olan="d">
    <variant olanPrefix="" aresti="1.1.2.1 (7)" name="Diagonal up line" correction="x">
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="5" group="POST" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
    </variant>
    <variant olanPrefix="i" aresti="1.1.2.3 (7)" name="Diagonal down line">
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MIN" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="5" group="POST" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="MAX" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
      <component pitch="ZERO" roll="ZERO" yaw="ZERO" length="1" group="NONE" />
    </variant>
  </manoeuvre>
</category>

```

*Figure 7: Example of the manoeuvre catalogue encoding scheme, the catalogue XML provided with OLANdroid is ~3,000 lines covering about 100 manoeuvres.*

An XML Schema Definition (XSD) is provided for the manoeuvre catalogue (Chapter 9.3), which the XML can be validated against, allowing any modifications to the catalogue to be checked before usage in the application, ensuring robustness both throughout development and with any further extensions. Figure 7 demonstrates some example manoeuvres and constituent components represented with the schema.

## b) Aresti Diagrams

The credit for the Aresti diagrams goes to OpenAero's SVG generation of Aresti diagrams from OLAN figures. The project, as its name suggests, is an openly licensed project (GNU general license), so taking this component is okay. The process was reverse engineered and diagrams extracted, scaled and rendered as PNGs with a script for use as image resources in this project.

## c) Texture

As one of the visualisation modes includes a textured surface as the ground, a resource for this is required. For this, a publicly licensed image was used.

## 4.3.5 Android Manifest

The Android manifest isn't really part of the design, but it plays an important role in the application, declaring essential information about the application to the Android system. For this project, it is used to declare a number of requirements and the hierarchy of



the GUI pages.

```
<uses-sdk
    android:minSdkVersion="11"
    android:targetSdkVersion="21" />

<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true" />

<uses-permission
    android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

*Figure 8: Section of the manifest file.*

First of all, as Figure 8 details, the manifest details the version of Android the project is aimed for – target and minimum SDK versions. This application supports all the way down to SDK version 11 (Android 3.0 – Honeycomb), and is aimed for version 21 (Android 5.0 – Lollipop). This covers a large range of Android versions, making the project accessible to about 95% of all Android devices out there [14]. Maintaining that the application worked on all of these versions involved the use of support libraries, which affected the design a little, but not greatly (which classes to extend and certain methods of doing UI-based things).

Another stipulation stated in the manifest for running the application is OpenGL ES version 2.0 (0x00020000). Android is now on OpenGL ES version 3.1, but devices which support that also support versions lower than that. Support for version 2.0 only became mandatory for devices in SDK version 14 (Android 4.0 – Ice Cream Sandwich), so there's a chance that the visualisation will not be available to devices between 11 and 14, but if the requirements stated in the manifest are not met, the application will not be available.

The manifest also declares requested permissions. To save flights, this application needs read and write access to a storage device as this is implemented with reading and writing a basic text file.

## 4.4 User Interface Design

The evolutionary design of the project also affected the user interface, with various iterations catering for different features and handling them in different ways. In the end, a user interface design emerged from utilising the user interface heavily during testing features (integration testing), so it has a focus on efficiency and convenience.

### 4.4.1 Structure

As previously mentioned, Android application user interfaces are split up into activities (pages) which consist of views (UI elements). These activities are built and divided

logically, to each enclose various parts of the application's operation, with a number of different activities for different aspects of the system:

- Flight manager (“FlightManagerActivity”) – options for creating new flights, renaming, editing, deleting and loading existing ones.
- Build a flight (“BuildFlightActivity”) – browsing the manoeuvre catalogue, inputting OLAN and editing the current flight.
- Visualisation (“VisualisationActivity”) – showing the visualisation of the flight, demonstrating the animation, moving it around in 3D space and saving the current flight.
- Settings (“SettingsActivity”) – changing various preferences.
- About (“AboutActivity”) – information about the project and its development.

The design also incorporates other user interface elements which are not directly part of an activity, including context menus, toasts (small unobtrusive messages) and dialogue boxes.

#### 4.4.2 Inspiration and Influence

Experience with using Android devices was useful for designing a user interface an Android user would expect – utilising common UX patterns like the menu bar, settings menu and heavy use of standard iconography. These are well honed designs, based on accessibility and some of the principals of user interface design [15]:

- Structure – activities have clear tasks and related things are kept together.
- Simplicity – only necessary features and UI elements are seen in the design.
- Visibility – only important, non-extraneous information is displayed.
- Feedback – changes to the state and condition of the application are clear.
- Tolerance – flexible navigation and allowing the correction of mistakes.
- Reuse – consistency in the design across various activities.

The look and interaction available with the user interface was considered to be a feature – usability and accessibility. This is important because some of the features offered by the application are quite complex, and the UI should make that as simple and accessible as possible as there is little point having features if they are so tricky to use or unclear that nobody uses them. The help button available on most activities aims to solve this problem further.

The main hardware influence on the user interface is the screen size and pixel density. Due to Android's system pixel scaling (density-independent pixels), which makes pixels correspond to a certain physical size, the density can more or less be ignored and the

screen size concentrated on. Screen sizes come in two main sizes – phones and tablets. It was decided early on to work with the same layout for both, instead of developing layouts for each separately. This made for a more refined, flexible layout, and one that works well with another consideration – screen rotation. Screenshot 1 and Screenshot 2 demonstrate user interface on two different form factors and orientations. It also shows the stylistic difference, a trait adopted from utilising the system “look and feel”.

### 4.4.3 Navigation

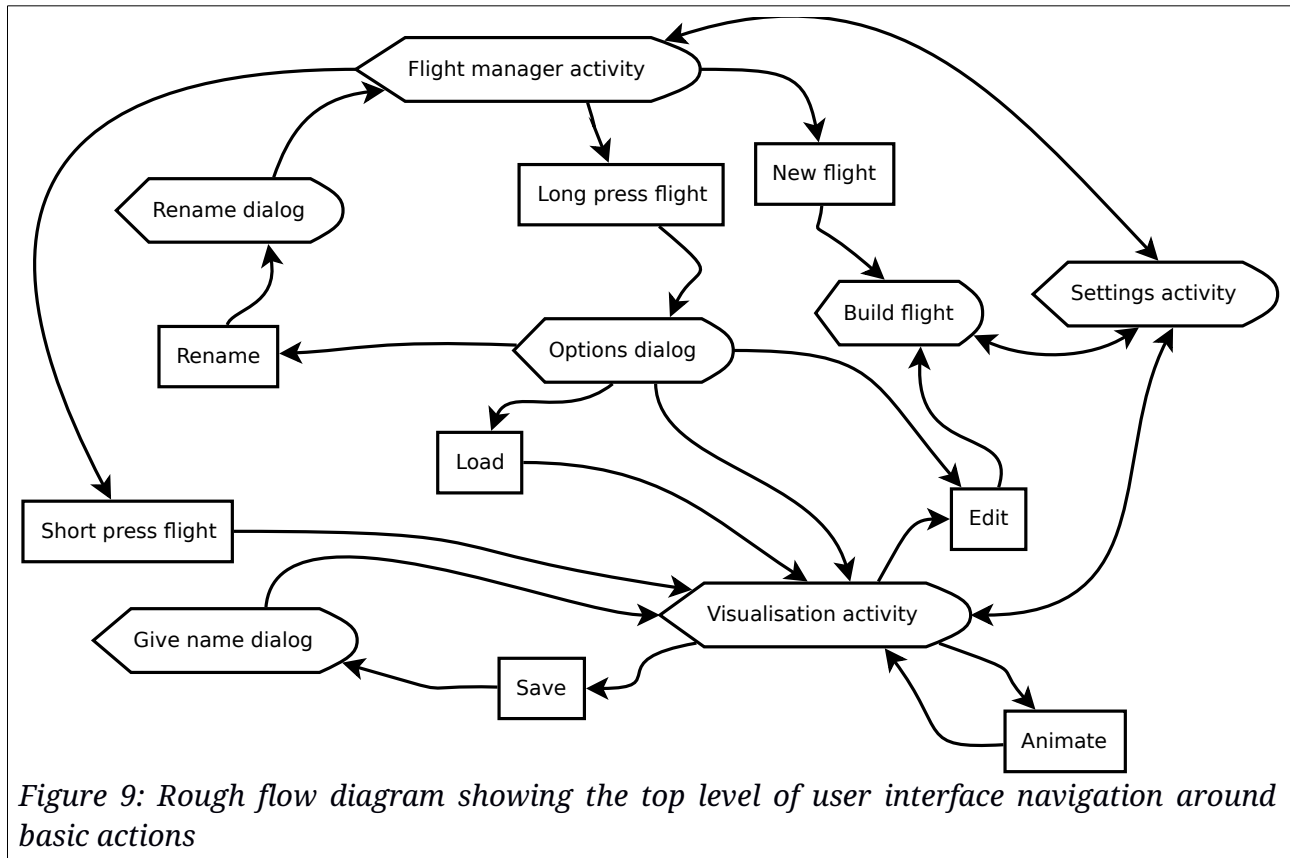


Figure 9: Rough flow diagram showing the top level of user interface navigation around basic actions

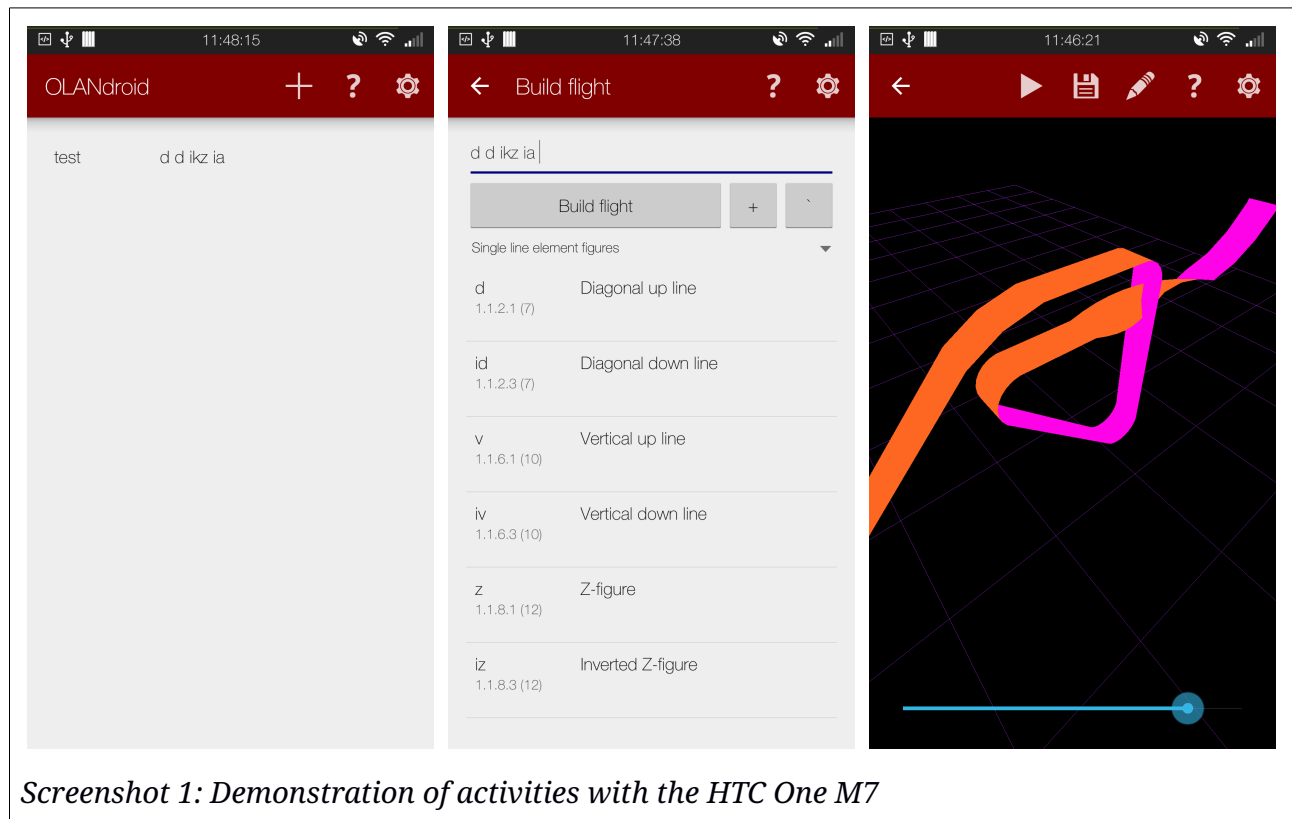
### 4.4.4 Style

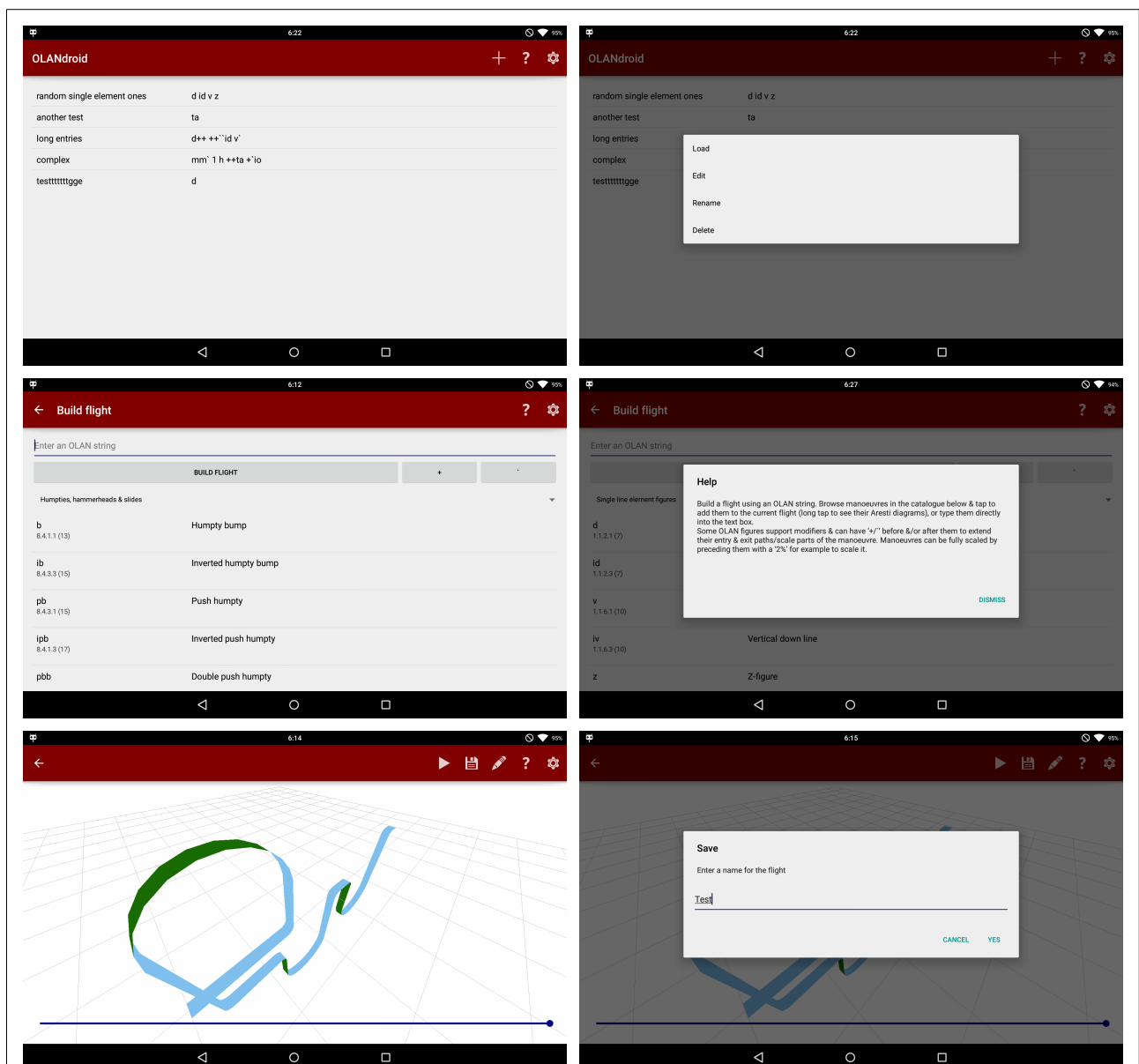
From a stylistic perspective, the UI is simple, with no distinct style of its own – making use of the system “look and feel” to look native. It uses a consistent colour scheme, making use of a red (RGB #800000) as its primary colour.

The visualisation has a style of its own, though it's kept fairly similar to that of the overall UI, using a simple colour palette of bright colours. Also offered in the visualisation is the choice colour palettes and ground styles, with a darker colour palette available for those

who prefer the light on dark look.

### 4.4.5 Examples





*Screenshot 2: Demonstration of activities and UI elements with the LG P-Pad 8.3*

# 5 Implementation

## 5.1 Code

Well documented, consistently styled code is an important part of what makes an application and its design comprehensible to developers. With this in mind, a rough style guide was adhered to, keeping code in check for readability.

All methods, unless inherited/implementing an interface, have their declarations commented simply using the Javadoc style to describe parameters, return types, exceptions thrown and more with general comments describing their functionality. This compiled Javadoc is provided with the source.

Android Studio also provides some static code analysis at the point of compilation and more on committing code through the Git integration. This was useful for improving code quality, so things like unnecessary imports, redundant methods, impossible loops and such like can be fixed/removed, making sure code quality is kept high.

Due to the evolutionary process of the application, the code, as the representation of the design, was constantly under review, making sure that it was the best for the feature set implemented. This also kept the code quality high.

## 5.2 Development Timeline

Summarising the development diaries gives a top-down view on how development progressed. This describes the progression of development as well as the design. The full development diaries are available as part of the source [7], and the weekly demos are available online as well [16].

### 5.2.1 Week one

Starting off the project after a meeting with the supervisor discussing the aerobatic flight and project goals. The week that followed involved mainly further research about aerobatic flight, the Aresti and OLAN systems and finding out how they work, as well as building up some ideas about to represent the manoeuvres. No real application code was written yet, as the development tools were being researched, organised and laid out for starting to develop on Android. Some of the research was focused on learning how to write for Android. A rough prototype of a visualisation of a cube was built using a tutorial to see

how things work.

### 5.2.2 Week two

By the second week, a way of representing the flight was settled on, with the basis for the component and manoeuvre classes using a forty five degree movement in the directions of pitch and yaw, and a length of one – unit vectors. Sequential drawing was set up with a matrix being developed and rudimentary manoeuvre with a simple line was drawn, with some 3D view manipulation. Also developed a recognition of manoeuvres as data so a prototype XML schema was built and work started on a related parser. Finished the project outline specification. Work so far on the project was demonstrated to the supervisor again, and met with positive feedback, and the decision that such frequent meetings were unnecessary.

### 5.2.3 Week three

The evolutionary design kicked in with force this week with lots of refactoring and the addition of a shape class to encapsulate OpenGL operations. Developed some thoughts on how to do the visualisation better – proving a ribbon instead of a line – and fix various bugs that had arisen. The user interface was revised to include an activity for reading in OLAN strings – not yet functional – and a small catalogue of parsed manoeuvres read in.

### 5.2.4 Week four

In the fourth week, tests were written for the already implemented features. The parser was expanded to be more dynamic, and the visualisation was modified to support drawing filled polygons rather than simple lines to better represent the manoeuvre. Later in the week, support for different colours for the top and bottom of the ribbon was added, making things clearer.

### 5.2.5 Week five

Started work on the animation with the addition of a method for drawing part of the flight/manoeuvre/component from zero to one and a seek bar to the visualisation for going through the animation. No manager for the animation yet as it worked through “OLANdroid”. Fixed a bug to do with two of the same manoeuvre in a flight breaking the animation due to them being references to the same object. Recognised a need for greater granularity in manoeuvres, so modified the components and the catalogue such that increments of fifteen degrees were used instead.

### 5.2.6 Week six

A lot of the work this week was spent encoding manoeuvres – the catalogue now being over two thousand lines. Support for rolls in components, entry/exit modifiers in the OLAN

interpretation and colour themes for the visualisation were added, as well as user testing rolled out with the application being added to Google Play with a small beta testing group of family members. More frequent Monkey testing began also.

### 5.2.7 Week seven

Another large user interface overhaul, with a more efficient navigation by removing unnecessary activities. Added support for the loading and saving of flights and another control scheme for the visualisation (two finger) inspired by user feedback. A big refactor of the animation code took place with it being separating it into its own managerial class. The name of the project was stumbled upon with the combination of OLAN and Android.

### 5.2.8 Week eight

The project saw its first real public demo in the eighth week, with it being demonstrated in front of the supervisor and various other people. Feedback suggested the need for help texts for unfamiliar users, so they were implemented. Support for the group scaling modifier using groups was implemented and the manoeuvres schema was modified to divide manoeuvres into categories.

### 5.2.9 Week nine

The flying wing animation style was developed, and option for it added. More manoeuvres appended to the catalogue.

### 5.2.10 Week ten

Recognised the visualisation as perhaps a bit too abstract, so added an alternative reference plane style – textured grass ground – which included redesign of the shaders and some additions to the shape. Developed a flight correction option to automatically add a relevant, corrective manoeuvre to prevent aircraft from flying into the ground (a problem which arose with a opaque textured ground). Fixed the animation for backward movement of components to support the added tailslide manoeuvres.

### 5.2.11 Week eleven

Found another OLAN modifier – full manoeuvre scaling – so implemented that into the manoeuvres and the OLAN interpretation. Development efforts were toned down from this week onwards to start writing this report.

### 5.2.12 Week twelve

Identified some anti-patterns in the code and worked to eliminate them by improving the cohesion of the controller classes and reducing coupling.



### 5.2.13 Week thirteen

Work on the report brought light to some possible developments, so Aresti diagrams were investigated, a method of extracting them from OpenAero was worked out, and they were added to the manoeuvre catalogue. Some of the UI needed to be improved to support the presence of them, so that was fixed also. The size of the user testing group was increased dramatically to garner more feedback.

### 5.2.14 Week fourteen

Feedback from the users taken into consideration, with the improvement of help texts and the limiting of view movement to inside the realm of the reference – so users couldn't get lost in the visualisation. Recognised issues to do with performance and large flights, so refactored the design so the manoeuvres were the drawable element (not the components) built from composing the vertices of their components, vastly improving performance.

### 5.2.15 Week fifteen

Went through the code to ensure strict compliance with coding style and generated the Javadoc. Took screenshots for this report, and spent the vast majority of time on this writing the report.

## 5.3 Finer Details

### 5.3.1 Component and Manoeuvre

One of the more interesting elements of the design is the “Component”, and how it represents a small section of a manoeuvre in both the model, and the visualisation.

For the visualisation, the component's role is to provide a “Drawable” object. It does this by constructing a set of vertices – points in space – which can be connected with triangles using a draw order to create surfaces – a basic fact of 3D modelling. These vertices are calculated from the four parameters of pitch, yaw, roll and length provided with the component and are relative to the origin (0, 0, 0). The X and Y axes are calculated using Euler angles (Figure 10), with the Z-axis considered to be forward movement and roll to be rotation on this line. Each component also has a related four-by-four matrix which defines the transformation between the start of the component and its end. This is calculated in much the same way as the vertices.

$$\begin{aligned}
 x &= \text{length} \times \sin(\theta) \times \cos(\varphi) \\
 y &= \text{length} \times \sin(\theta) \times \sin(\varphi) \\
 z &= \text{length} \times \cos(\theta) \\
 \text{where } \theta, \varphi &\in \left\{ \frac{-1}{24}, 0, \frac{1}{24} \right\}
 \end{aligned}$$

*Figure 10: Euler angles for a component, the set defines the bounds as a fraction of a full circle and needs to be converted into angles using either radians or degrees.*

As each component is assembled from the origin, they can be drawn sequentially to build a manoeuvre – multiplying each vertex by the cumulative transform of all of the previous components. In practice, it's not done quite like this as this is highly inefficient. Instead, the drawing stems from the “Manoeuvre”, with the vertices of all the components in a manoeuvre are appended to one buffer, each multiplied by the cumulative transform of all preceding components. This kind of acts like a cache, and needs to be recomputed for any animation within the manoeuvre or the base matrix changing, but it still provides performance benefits.

### 5.3.2 Animation

Animating the flying wing animation style of flight is a fairly complex problem. The “Drawable.Flight” interface describes the animation function as taking two values between zero and one, denoting the animation progression of the front and the back of flight piece. The algorithm uses the total length of the flight, the length of the flying wing and the cumulative length of manoeuvres to work out its two values, which are then passed off to the right manoeuvres, which carry out a very similar process again but with components instead, and requests a rebuild of its vertices cache. The component takes the values and scales its vertices (the ones at its origin and/or the further ones and this is drawn.

### 5.3.3 User interface

One of the tenets of the MVC is the separation between the components. With this in mind, the code in the activities (view) is such that it only calls the controller code, and does not do any operation itself.

It should be noted that the size of the “SurfaceView” UI element – the canvas which the visualisation is drawn on – is set to fill the entire activity it is in (other than the action bar). This means that the more pixels on the device screen, the more pixels the shaders have to poll, which can mean increased demands on the hardware.

## 5.4 Issues

A few issues were ran into during the implementation phase of the project. A usable

solution or a workaround was found for majority of these, but for some this meant a compromise of design.

One of the main problems with the design is the demand for a “Context” object for accessing file resources. This constrained the design a bit and tied parts of the application together which don't really belong near each other. UI components have simple access to contexts, but at a lower level, in the renderer and model parts for drawing where access to files for shaders and textures are needed, there is no way of accessing a context.

An issue with the animation was ran into in the case of having flights with multiple of the same manoeuvre – both manoeuvres, no matter where they were in the flight, animated in unison. It turned out that this was due to the fact that, when querying the manoeuvre catalogue, a reference to an instance of a manoeuvre is returned, thus these two are the same for when a flight has two of the same manoeuvre. This was fixed by implementing a method for cloning manoeuvres on retrieval.

Another problem came about through the animation. The progress of animation should be between zero and one, and when the animation thread runs, it iterates the progress through between those values about in thousandth increments (quite low, but it ensures smoothness on high-end devices). The value is represented as a float in Java, and incrementing the value like this accumulates inaccuracies of the number representation, causing crashes when the value was over one. Switching it to a multiplication instead of an incrementation fixed this issue.

One tricky issue that was come across with the user interface was the flights list not updating to reflect new flights being added. This works via a list adapter, which listens to an object for changes and repopulates the UI list on noticing changes. Originally, the method for getting a list of flights from the “FlightManager” converted its dynamic, “ArrayList<Flight>” into an array, which meant that it had a different memory address every time it were called, so the list adapter was listening to the wrong thing. This was solved by simply returning the dynamic list object instead of an array.

Manipulating a 3D world with a limited set of controls – just a touch screen – is difficult. Various other Android applications which tackle the same problem were looked into, and inspiration taken from these, but the solution implemented is still not ideal, which is why two different control schemes are available. Getting these control schemes to feel natural was also difficult, so a scaling factor was implemented, alongside another factor which takes into account the dimensions of the screen, to give a more natural feeling to the movement.

# 6 Testing

Testing is an important part of any software development. It is useful for ensuring consistent and expected action across a range of scenarios, and eliminating problems in the code.

A lot of testing for this project was white box continual integration testing achieved through the development process of changing a piece of code, compiling, building the application, pushing it to the device and then testing it then and there. With builds being automated with Gradle and a fair powerful, this process takes no longer than ten seconds, and even less time when using the virtual device, allowing for a rapid development and testing cycle. This type of testing was useful for identifying and fixing problems as soon as they arose, but it was not a formal process by any means.

Some components call for a more rigid, orderly testing mechanism though, such as the OLAN interpretation. For parts like this, automated unit tests were written which could experiment with as many situations and inputs as possible. Running these frequently was crucial for maintaining robustness. The stress testing via Monkey also forms part of this automated testing.

However, on the other side of things, parts of the application such as the visualisation cannot be tested with formal tests, due to their visual nature. Their responsiveness and resilience to crashing can be exercised formally – stress testing – but these testing this part of the application is mainly a qualitative affair.

## 6.1 Device testing

A number of different physical devices were used for the testing, as well as an emulator for testing a wider range of API levels. Physical device testing was useful for testing it out in the field and how well touch controls work etc. As mentioned earlier, there is quite a wide range of Android hardware available, and while the devices at hand were not representative of that full spectrum, they still were of use to build up a picture of how the differences in software affect the project.

Device	HTC Sensation XE	HTC One M7	LG G-Pad 8.3
Year	04/2011	02/2013	10/2013
Android version	4.2.2	4.4.3	5.0.2
API version	17	19	21
Android ROM, UI skin	CyanogenMod 10, Holo	ViperOne 7, Sense 6	CyanogenMod 12, Material
Screen size	4.3"	4.7"	8.3"
Resolution	540x960	1080x1920	1920x1200
Processor	Dual-core, 1.2GHz	Quad core, 1.7GHz	Quad core, 1.7GHz
RAM	768MB	2GB	2GB
Graphics	GPU	GPU	Integrated

*Figure 11: table of the main physical devices used for testing [18]*

User testing opened up the number of different devices tested on. Though they were of similar hardware, software versions – particularly Android OEM modifications – had an influence (though results on UX were positive).

Alongside physical devices, the Android SDK provides an emulator. This a useful tool for testing as well, as the version of Android it has is very easily changeable, so the full range of the API levels this application is aimed at can be tested. Different SDK levels also have access to different versions of Java as well as the Android interfaces so there are several areas of the code which are exercised by this testing.

## 6.2 Unit Tests

The unit tests mainly tested the controller parts of the project, though they exercised the methods available in two different manners – through the user interface's controls and natural feedback, as well by direct manipulation. This allowed the unit tests to cover both black and white box testing. They are written making use of the popular unit testing library JUnit and Android-specific testing classes.

As the visualisation is difficult to test with unit tests, the tests are mainly focused around the flight manager with its OLAN interpretation and file saving/loading. Tests also ensure that navigation works as expected and UI pages are constructed correctly with all of the important elements present.

Tests were written throughout the project, though not directly tied to any point in the process. A description of various unit tests can be found in Chapter 9.4. Most of the tests are

focused on one part of the system, but some overflow into exercise other components, such as the UI-based interaction tests which use the controller for ground truths. Generally, these tests run after checking the controller separately, making sure it provides valid responses. The vast majority of unit tests are data dependent – primarily relying on manoeuvres being available, this does mean that some static data needs to be changed in the test classes occasionally to reflect the state of the data.

Generally, the unit tests gave a pretty good impression of how well the system was working, and they were valuable for finding any bugs which arose during refactors, particularly the tests which exercised the operation through the UI. At any released version of the application, it was made sure that all of the tests that had been written up to that point were passing.

## 6.3 Stress Testing

There are three main components to the application which need stress testing – the user interface, the model and the visualisation.

First, the user interface can be stress tested. One of the best ways of doing this is using the Monkey test tool [17] (Figure 12). This is a program which runs on a testing device/emulator and stresses the user interface by generating a quantity of random user events, such as touches, clicks and gestures, as well as system-level events which might affect the application. It is simple to run, and can be configured to generate thousands of events per run, thoroughly exercising the UI. Any problems which are found can be tested again by running the tool again with the same seed.

```
[gideon@computer]:[~/android/sdk/tools] adb shell monkey -p uk.ac.aber.gij2.mmp -v 5000
{lots more events}
:Sending Trackball (ACTION_UP): 0:(0.0,0.0)
:Sending Trackball (ACTION_MOVE): 0:(-4.0,-1.0)
//[calendar_time:2015-04-20 16:45:50.298 system_uptime:679342286]
// Sending event #4900
:Sending Touch (ACTION_DOWN): 0:(1038.0,1191.0)
:Sending Touch (ACTION_UP): 0:(1024.404,1212.2122)
:Switch:
#Intent;action=android.intent.action.MAIN;category=android.intent.category.LAUNCHER;launchFlags=0x
10200000;component=uk.ac.aber.gij2.mmp/uk.ac.aber.gij2.olandroid.view.FlightManagerActivity;end
:Sending Touch (ACTION_DOWN): 0:(57.0,282.0)
:Sending Touch (ACTION_UP): 0:(59.05987,226.81026)
:Sending Trackball (ACTION_MOVE): 0:(2.0,-1.0)
:Sending Touch (ACTION_DOWN): 0:(202.0,195.0)
:Sending Touch (ACTION_UP): 0:(200.10712,190.59319)
Events injected: 5000
:Sending rotation degree=0, persist=false
:Dropped: keys=11 pointers=1 trackballs=0 flips=0 rotations=0
## Network stats: elapsed time=30690ms (0ms mobile, 0ms wifi, 30690ms not connected)
// Monkey finished
```

*Figure 12: a small piece of example monkey output and conclusion*

Any time a large change was made to the user interface, Monkey was used to test, alongside the usual integration testing throughout development. The UI was exercised such that as many different cases as possible were found and used, to make sure that any edge cases and related issues were discovered and the problems ruled out.

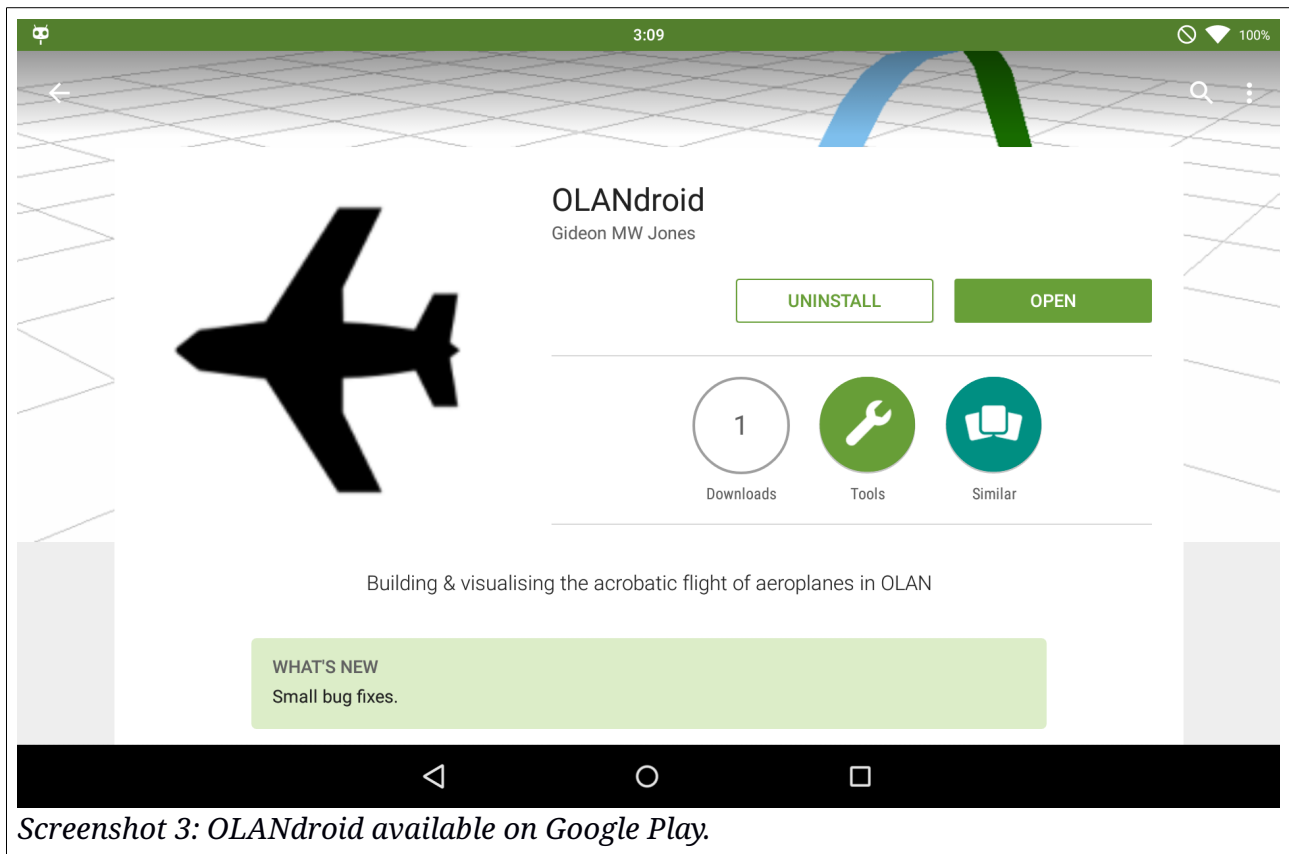
The model of the flight, and resultantly the OLAN interpretation is stress tested with using a wide range of different constructs of flights. This tests the robustness of the input parsing and how well the model can cope with complex/long flights. Unit tests are written for this, using a range of various different complicated OLAN sequences. Generally, the model can cope with very large flights, due to the only real limitation to this being the RAM available.

Finally, the visualisation is a component of the project which needs stress testing as well, as it's the most performance-demanding component. This was done using a similar process to the model testing, with a range of flights entered, built and experimented with. While the model can cope with very large flights, the visualisation can struggle with them, with there being a significant slowdown with flights of over forty figures, most noticeable during the movement of the view. As mentioned before, the ability to cope with these is dependent on the hardware available.

## 6.4 User Testing

Black box user testing is a great way to expose issues with the application. As the sole developer and user, any decisions made when testing the application are influenced by an understanding of the system which can mean inadvertently avoiding potential issues and using the application in patterns that every day users might not do, making tests unfair. Using a user group unfamiliar with the application is more representative of how “real users” interact with the project, and generates more useful feedback.

To be able to effectively allow outside users to test the application, a distribution platform is needed. This provides any testing users with some reassurances, such as a lower likelihood for the code to be malicious – distribution platforms tend to scan and authorise applications – and a simplicity of installation/removal – many users do not understand how/are not comfortable with manually installing applications on mobile devices. It also provides some benefits to the developer too, with release management, limiting the user base and analytics such as amount of downloads and App Not Responding messages (ANRs)/crash reports.



*Screenshot 3: OLANdroid available on Google Play.*

Google Play was chosen (Screenshot 3) due to its ubiquity on Android, its closed beta testing scheme and analytics. Several other students and family members got a chance to download and try out the application, and feedback was discussed in a Facebook group to manage it all. Every one of the testers was unfamiliar with aerobatic flying, so it was an ideal group to test with as it matched the target audience well. In this way, these user tests can be considered acceptance tests, as the goal is to be able to help these people understand the aerobatics.

Over the course of user testing – which lasted about seven weeks – twenty five successive versions of the application were available to users. In terms of stability, no crash/ANRs were submitted across all versions, demonstrating the effectiveness of the testing used. Feedback on builds of the application from users during development was increasingly positive, with suggestions which led to improved UI, control schemes, help dialogue texts and other features, resulting in a better, more usable application.



# 7 Critical Evaluation

With regard to the original goals set out for the project, the result does fairly well. It satisfies the main points of the brief in an effective, accessible manner, with the final application being able to parse an input flight description, build it and display it in a 3D environment, allowing users to animate and move around it to get a feel for the flight. It also offers users a catalogue of manoeuvres with related Aresti diagrams and catalogue references, which makes building flights easy for those with completely no knowledge on the subject, and helps them develop ties between the manoeuvres and the Aresti diagrams. Alongside all of this, it even provides features designed to make the application easy to use, with allowing users save flights and edit flights, and offering a range of settings for users to customise their experience through various visualisation aesthetic and usability options.

## 7.1 Research and Analysis

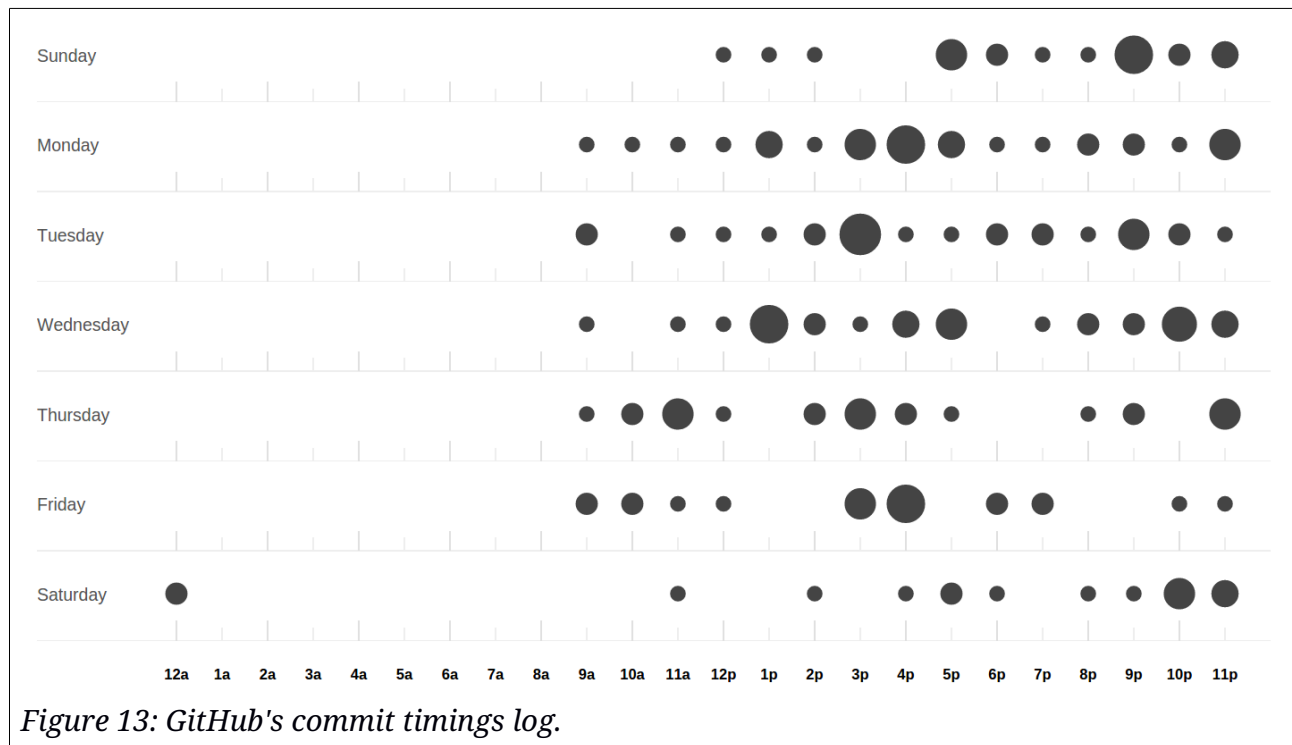
The research carried out in the starting stages of the project was informative and helped greatly develop ideas about the project. However, it was quite weighted towards investigating how to design the model. More research which engaged with actual model aircraft fliers would have been hugely useful. Considering their experience of the topic, their expertise and advice as to what is useful in an application such as this could have focused the project a bit further.

With concern to the analysis, perhaps a step further back would have benefited the project more. This would have allowed for a greater number of manoeuvres to be considered when thinking of ideas for the design, and perhaps resulted in a better design, rather than running ahead with a potentially non-optimal design.

## 7.2 Process

The process used for developing this project worked fairly well. It turned out to be advantageous to not be locked into a strict methodological framework, due to the inexperience with the technologies in use and, to a degree, inexperience with maintaining a strict methodology. Originally, there was a worry that this lack of formality in the approach taken, when compared to rigid methodologies like waterfall or scrum, would negatively affect the development, but work proceeded at a fairly consistent pace throughout. An

exception to this is the end of the project, where work slowed down not due to lack of work – there's a number of possible extensions – but due to this report.



The decision to go for a flexible, agile process was the right one. As commits were only made for notable code and feature changes, Figure 13 demonstrates how a flexible the timings of work needed to complete working features be due to unfamiliarity with the platform.

Feature driven development was a sensible inspiration for the methodology used. It didn't stand in the way of progress with the bureaucracy/demands of comprehensive documentation, which some methodologies can do, and instead followed the ideal, natural rhythm of development. A rough feature list could be gleaned from the project brief and the planning phase of FDD manifested itself in the analysis when parts of the design became apparent. The iteration also similarly matched the one used in the end, but without the dedicated up front planning bit to each iteration, instead a design evolved throughout the iteration from prototype to tidy, refactored, complete feature.

During development, the true value of the practice of merciless refactoring for simple design was realised. It should always be noted that merciless refactoring requires an element of bravery – to break things apart completely and rebuild in such a different way. If this epiphany was had a bit earlier, perhaps a more effective design could have been reached a bit earlier.

One aspect that the process did not fulfil as much was hoped was the testing. Placing a

value working code over comprehensive testing seems a bit paradoxical, but since things were tested so frequently during writing the code, it didn't seem necessary to spend time that could be spent developing on writing tests. Ideally, unit testing would have played a greater role in the iteration, perhaps even as far as going for a test-driven-development model utilising the red/green/refactor system for writing code.

## 7.3 Design

The resulting design of the application is fairly robust, but it's not perfect. The depth of the analysis and the lack of experience with both the platform and graphic applications in general are an influence here, but on the positive side it does make good use of the facilities of object-orientation and design patterns.

Where the design succeeds well is the separation from the controller and, to a degree, from the view. It maintains an observable-type passive model, meaning that it is more robust to changes in the user interface and the controller interacts with it.

Due to not having a formal manoeuvre catalogue at hand, such as the Aresti one, it was (perhaps ironically) difficult to build up a knowledge of manoeuvres. In the research and analysis phase, having these manoeuvres was crucial for developing a design for their internal representation. In the end, the shapes of a limited set of manoeuvres was taken from descriptions of incomplete sets of available manoeuvres in OLAN (program) and OpenAero. The model of components and their motion comes from the highest common factor of movement in manoeuvres, and since the set was incomplete, it is the case that the representation used in this project cannot support every manoeuvre. Manoeuvres which rely on momentum-based movement – like flick-rolls – are not supported, and neither are those which require movement on two axes at once – such as a rolling curve. These are shortcomings in the design, and not one ones easily solved. A solution to the latter would be the allowance of free-reign of movement with components – allowing any degree of axial movement – but that potentially sacrifices the realism of the manoeuvres model, and makes encoding them far harder. The model is thus a bit of a compromise, however all is not lost as the set of manoeuvres possible to encode in the application is still fairly large.

There is the worry that the singleton design pattern is overused in the application. It seems clear that some components of the design should be limited to only one copy for maintaining state consistently, but utilising a singleton for all the managerial classes is perhaps overkill.

“Permission creep”, the requesting of unnecessary permissions by applications, is considered to be a problem on Android [20]. While project only is pretty good and only

requests one permission for file reading/writing, this permission is kind of scary as it allows access to much more than a user would expect. It could not request any permissions at all, with saving flights in a similar way to preferences. This is ugly though, and not a proper usage of the facilities provided in the SDK, so the decision made to use this permission was the correct one, and a the result of much consideration.

## 7.4 Features and Extension

The final application resulting from this project is fairly feature complete. In fact, it has more features than was initially planned in the construction of the feature list. This begs the question of whether the project was initially ambitious enough. Perhaps work could have been made towards the stretch goal of building a connection between the flight itself and the controls necessary for recreating the flight in question, but in the end, this was not tackled in the project due to a higher priority being placed on building a robust application which fulfilled the main brief. Given a bit more time, it is reasonable to assume that it could be done – with obvious ties being made between the components' principal movements and the stick movements – but likely not finished to the level of polish available in the rest of the application, so it was put aside.

A lack of a formal specification for OLAN made building a complete parser hard. The modifiers available in this application are only ones derived from example flights available in OpenAero. Users more familiar with OLAN might have difficulty with this aspect of the application then, as it might not be as complete as thought.

Acknowledging the importance and dominance of the Aresti system in this field suggests that the support for the system is perhaps a little lacking in the project. Efforts were made to provide some connection between the OLAN and Aresti systems through including Aresti diagrams and references in the catalogue, but they're a bit hidden away and not as well explained as they could perhaps be. This is an area which could definitely be improved, with the UI being a prime candidate for work.

Opening up the manoeuvre catalogue, allowing users to define their own manoeuvres, would be a great way of increasing the openness of the application. It's difficult to think of an easy way to do this, but it would be a potentially useful feature to users. As this project is an open-source one, the code and the data is available on GitHub [7] which allows users to fork it, create their own modifications and send a pull request – asking that their modifications become part of the main source. This opens up the manoeuvre catalogue a bit.

## 7.5 Platform and Tools

Developing on Android turned out to be fairly simple, thankfully. Knowledge of Java was useful, as well as familiarity with OpenGL – thanks to how similar the library is to the WebGL bindings – meant that development could progress quickly. The opportunity provided by this project to learn Android was relished in also, and definitely something which will be of use in the future due to a developed interest.

## 7.6 Conclusion

The application developed throughout this process manages to do the job of educating people unfamiliar with aerobatic manoeuvres pretty well. It's not perfect, with a number of manoeuvres not available and a need for refining the model (“A poem is never complete, only abandoned” [21]); and nor is it fully complete, with there as always being scope for further extension; but it still provides a useful, robust experience, and that is a satisfactory result.

# 8 Annotated Bibliography

- [1] Ringo Massa, Wouter Liefting, Gilles Guillemard, Christian Falck, “OpenAero”, [online]. Available: <http://www.openaero.net> [Accessed: 2015-01-30].  
OpenAero was an indispensable tool for learning about the OLAN system, the catalogue of manoeuvres available and the modifiers. It also providing Aresti diagrams were useful for constructing flights using components as well, building up a 3D picture of how the aircraft moved.
- [2] Michael Golan, “OLAN”, [online]. Available: <http://olan-2011.software.informer.com/> [Accessed 2015-01-30].  
The OLAN application proved useful with its similar functionality to OpenAero, except this tool provided a more definitive explanation of the OLAN system and a catalogue describing all of the OLAN figures available – useful for building a catalogue for this application.
- [3] Thomas Halleck, “We Spend More Time On Smartphones Than Traditional PCs: Nielsen”, *International Business Times* [online], 2014. Available: <http://www.ibtimes.com/we-spend-more-time-smartphones-traditional-pcs-nielsen-1557807> [Accessed 2015-02-15].  
An interesting article which reinforced the decision to choose a mobile platform over a PC, offering even more incentive to develop for mobile.
- [4] “IDC: Smartphone OS Market Share,” *International Data Corporation*, 2014, [online]. Available: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp> [Accessed: 2015-01-30].  
Data from a respected company demonstrating the dominance of Android in the mobile space, and suggesting it to be the right platform to develop for to get the widest reach of the application.
- [5] “RC Groups,” *Model aircraft enthusiasts forum*, [online]. Available: <http://www.rcgroups.com/forums/index.php> [Accessed: 2015-02-10].  
A forum frequented by model aircraft flying enthusiasts was a good place to look for opinion on the OLAN and Aresti formats, as well as developing an idea about what sort of tools were available for teaching newcomers to the hobby how to fly manoeuvres.
- [6] Stephen Palmer and John Felsing, *A Practical Guide to Feature-Driven Development*, 1st ed. Prentice Hall, 2002.  
A useful introduction into a methodology which, while not strictly followed, was definitely an influence to the process used. It highlights the strengths and weaknesses of the methodology and its suitability for various sorts projects.

- [7] “The OLANdroid project source on GitHub”, *GitHub*, [online]. Available: <https://github.com/GideonPARANOID/olandroid> [Accessed: 2015-01-30].  
GitHub was used as one of the remote storage places for this project via the Git version controls system. Alongside simply storing the code, it provided analytical data with regard to when commits happened and the quantity of code changed across time, giving a way of measuring progress and seeing the effectiveness of the methodology used.
- [8] Oracle, “Javadoc documentation and description of the system”, *Oracle*, [online]. Available: <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> [Accessed: 2015-03-10].  
Useful documentation with regard to specific syntax of various parts of the Javadoc system, so the parser did not have any trouble when building the documentation from the source. Also provided a recognisable, consistent style which would be familiar to other developers, making the code simpler to understand.
- [9] Kevin Brothaler, “Learn OpenGL ES”, [online]. Available: <http://www.learnopengles.com> [Accessed: 2015-02-01].  
A site offering extensive tutorials on how to use OpenGL ES 2.0 with Android, with plenty of examples and explanations. This was instrumental in understanding how the interface worked and for building the renderer and drawable components of the application.
- [10] Tek Eye, “List of IDEs for Android App Development, Which is Best for You?”, [online]. <http://tekeye.biz/2014/list-of-android-app-development-ides> [Accessed: 2015-02-01].  
A useful rundown of various IDEs, explaining the positive and negative aspects of each and covering the difference between ADT-based (Android Development Toolkit) IDEs and Google's Android Studio.
- [11] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison Wesley, 1994.  
A definitive guide on the topic of software design patterns, detailing many of the patterns used to solve problems in the design of the application, including observer/observable, singleton and model-view-controller.
- [12] “Futurice compiled Android best practices on GitHub,” *GitHub*. [online]. Available: <https://github.com/futurice/android-best-practices> [Accessed: 2015-02-06].  
A open project consisting of the compiled knowledge of Futurice – a Finnish mobile software company – developers, giving a useful insight into good practices for Android development from years of experience. Useful for steering the design in a sensible direction when previous experience on the platform was non-existent.
- [13] “Java - Singletons versus Application Context in Android?”, *Stack Overflow*, [online]. Available: <http://stackoverflow.com/questions/3826905/singletons-vs-application-context-in-android> [Accessed: 2015-02-14].  
Discussion with regards to whether or not to subtype the Application class in Android



or to use a singleton, including responses from an Android framework engineer. This provided some positive and negative aspects of various different implementations of one of the most important classes in Android development, and influenced the design a fair bit.

- [14] “Google Android dashboard - software version/hardware distribution,” *Google*, [online]. Available: <https://developer.android.com/about/dashboards/index.html> [Accessed: 2015-03-05].

A set of data obtained from Google Play usage providing some valuable insight into what versions of software, size screens etc. are commonly found across Android devices, helping focus development efforts to support those devices. This data also played a significant role in deciding what SDK level to pitch the application at.

- [15] Larry L Constantine, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, 1st ed. Addison Wesley, 1999.

A book describing some of key principals of user interface design, giving focus to what components of the UI need the most work, and how to build a highly usable, accessible application.

- [16] “University work”, *Gideon MW Jones on YouTube*, [online]. Available: <https://www.youtube.com/watch?v=22nCymgtDWk&list> [Accessed: 2015-01-26].

A series of screen captures from most week's of development, similar to the weekly development diaries, these capture the progression made through development.

- [17] “Monkey UI test tool,” *Google*, [online]. Available: <https://developer.android.com/tools/help/monkey.html> [Accessed: 2015-02-20].

Description and documentation on the automatic-event-generating Monkey test tool used for stress testing.

- [18] “Phone news, specs and reviews,” *Phone arena*. [Online]. Available: <http://www.phonearena.com/>. [Accessed: 20-Apr-2015].

Data on the technical specs of the test devices, useful for comparing them and the application experience they provide with relation to their hardware, and, to a degree, extrapolating how other devices would perform.

- [19] Lana Wachowski and Andrew Wachowski, *The Matrix*. 1999.

Inspiration on the complete free-reign afforded by a virtual environment, in terms of visualisation and further, and how it is important that that aspect not stray too far from reality because it could be rejected by users (as confusing).

- [20] Timothy Vidas, Nicholas Christin and Lorrie F Taylor, *Curbing Android Permission Creep*, IEEE Computer Society's Technical Committee on Security and Privacy, 2011, [online]. Available: <http://www.ieee-security.org/TC/W2SP/2011/papers/curbingPermissionCreep.pdf>

Comment on the issue of Android permission creep, and how it presents a problem for the platform.



- [21] Paul Valéry, paraphrased by W H Auden, *Collected Short Poems*, 1965.  
An observation on the in completion of poems, found relative to computer programs also.

## 9 Appendices

### 9.1 Libraries and Tools

#### 9.1.1 Android SDK

The Android SDK, developed primarily by Google, provides a wide range of tools and code libraries for the use of Android development. Including ADB – a tool for managing Android packages and system via USB/network; a device emulator with support for a wide number of SDK versions; Monkey – a tool for stress testing user interfaces; and much more.

#### 9.1.2 Android Studio and Gradle

The IDE chosen, Android Studio, provided a lot of tools, such as Git and Gradle integration; Android SDK tools integration, like ADB, simple access to a virtual device, memory monitoring; code completion, refactoring tools, UI design tools and more. It was indispensable for development, making progress a lot faster than if all the tools were used manually.

#### 9.1.3 OpenAero

A vital tool for understanding the OLAN system and building up a catalogue of manoeuvre was OpenAero. Due to its open-source nature, it was useful to be able to examine the web app's code and derive the SVGs for the Aresti diagrams used in the application.

#### 9.1.4 Git, GitHub and Bitbucket

Git served as the version control system used to track changes across the project's code, resources and report. It was used in conjunction with two remotes – GitHub and Bitbucket – which stored copies of the code. Two were used for redundancy's sake and because GitHub suffered downtime during development (due to a DDoS attack).

#### 9.1.5 Dia, LibreOffice Writer and Zotero

These first two tools were used for building diagrams and the general formatting of

this report. The latter was useful for maintaining references and keeping track of useful pages visited, via the Chromium (web browser) extension and LibreOffice integration.

## 9.2 Glossary

Term	Definition
ADB	Android Debug Bridge
ANR	Application Not Responding
API	Application Program Interface
DDoS	Distributed Denial of Service
IDE	Integrated Development Environment
MVC	Model View Controller
OEM	Original Equipment Manufacturer
OLAN	One Letter Aerobatic Notation
ROM	Read Only Memory
SDK	Software Development Kit
SVG	Scalable Vector Graphic

## 9.3 Manoeuvre Catalogue XSD

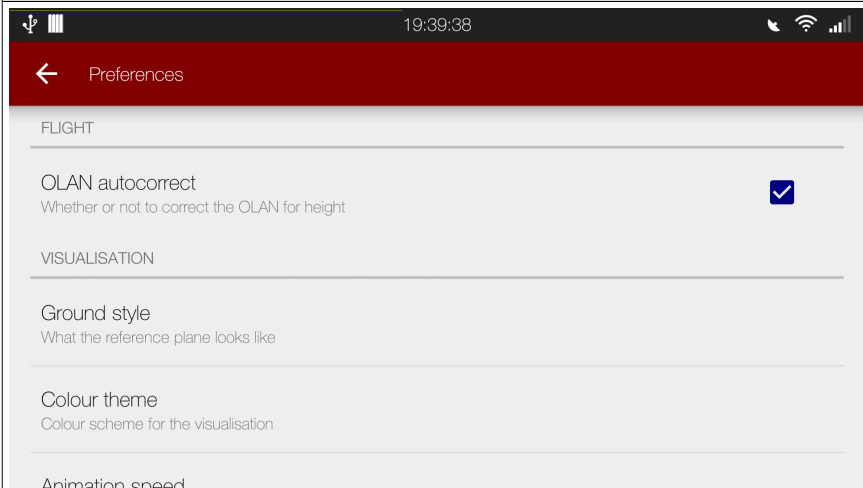
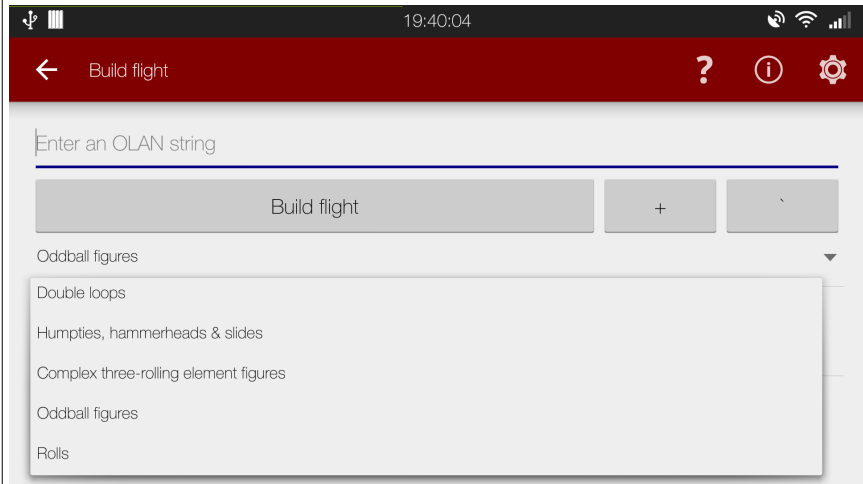

```
<xs:schema
  attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="catalogue">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="category" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="manoeuvre" maxOccurs="unbounded" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="variant" maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="component" maxOccurs="unbounded" minOccurs="0">
                            <xs:complexType>
                              <xs:simpleContent>
                                <xs:extension base="xs:string">
                                  <xs:attribute type="xs:string" name="pitch" use="required"/>
                                  <xs:attribute type="xs:string" name="roll" use="required"/>
                                  <xs:attribute type="xs:string" name="yaw" use="required"/>
                                  <xs:attribute type="xs:float" name="length" use="required"/>
                                  <xs:attribute type="xs:string" name="group" use="required"/>
                                </xs:extension>
                              </xs:simpleContent>
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

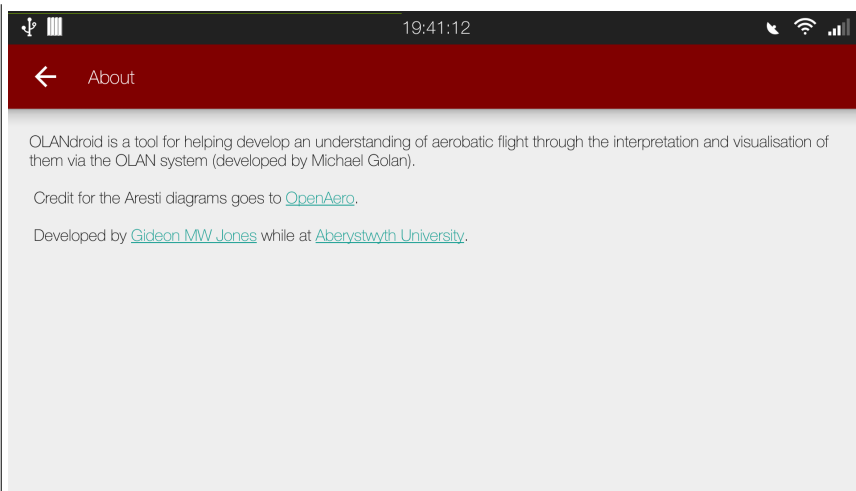
## 9.4 Unit Testing table

Component	Name	Description
FlightManagerActivity	testBuildUI	Ensures the UI is built correctly with all items present
FlightManager and FlightManagerActivity	testFlightList	Ensures the flight list is suitably populated
FlightMangerActivity	testNewFlightButton	Ensures the new flight button starts the right activity
FlightManager	testAddFlight	Ensures a new flight is added

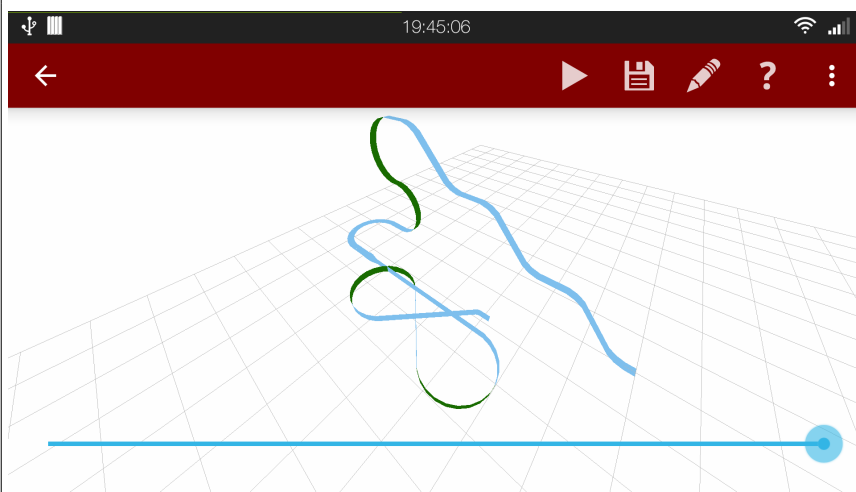
FlightManager	testDeleteFlight	Ensures a flight is deleted
FlightManager and ManoeuvreCatalogue	testValidOLAN	Ensures flights are created
FlightManager and ManoeuvreCatalogue	testInvalidOLAN	Ensures flights are not created
FlightManager	testSaveAndLoad	Ensures all of the flights are saved and loaded
FlightManager and ManoeuvreCatalogue	testNewFlightCorrection	Ensures flights are corrected suitably
BuildFlightActivity	testBuildUI	Ensures the UI is built correctly with all items present
BuildFlightActivity	testPlusButton	Ensures the modifier insertion button works
BuildFlightActivity	testBacktickButton	Ensures the modifier insertion button works
BuildFlightActivity and ManoeuvreCatalogue	testCategorySpinner	Ensures the category spinner is suitably populated
BuildFlightActivity and ManoeuvreCatalogue	testManoeuvreList	Ensures the manoeuvre list is suitably populated
BuildFlightActivity and ManoeuvreCatalogue	testValidOLAN	Ensures building a valid flight through the UI works
BuildFlightActivity and ManoeuvreCatalogue	testInvalidOLAN	Ensures building an invalid flight through the UI doesn't work
ManoeuvreCatalogue	testGetManoeuvre	Ensures the catalogue has loaded
ManoeuvreCatalogue	testCategories	Ensures all of the categories are loaded
ManoeuvreCatalogue	testCorrectionManoeuvre	Ensures the correct correction manoeuvre is loaded
VisualisationActivity	testBuildUI	Ensures the UI is built correctly with all items present
VisualisationActivity and AnimationManager	testAnimationSeekBar	Ensures that moving the seek bar sets the animation progress
VisualisationActivity and AnimationManager	testSetAnimation	Ensures that the UI reflects the animation's progress
VisualisationActivity and AnimationManager	testPlayAnimation	Ensures that the animation progresses on playing
VisualisationActivity and AnimationManager	testPauseAnimation	Ensures that the animation pauses after playing

# 9.5 Further Device Screenshots

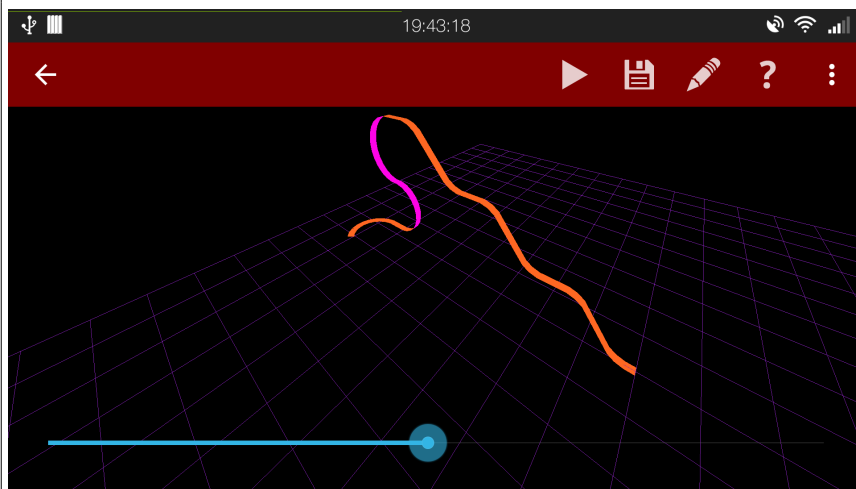
Screenshot	Description
	The settings menu, with various settings grouped by topic. The “SetttingsActivity” makes use of two different types of options – checkboxes and multiple choice lists
	The spinner in “BuildFlightActivity” showing off the various different categories of manoeuvres loaded from the catalogue file and available for usage in built flights
	An Aresti diagram (from OpenAero) brought up in “BuildFlightActivity” by long-pressing a manoeuvre in the catalogue



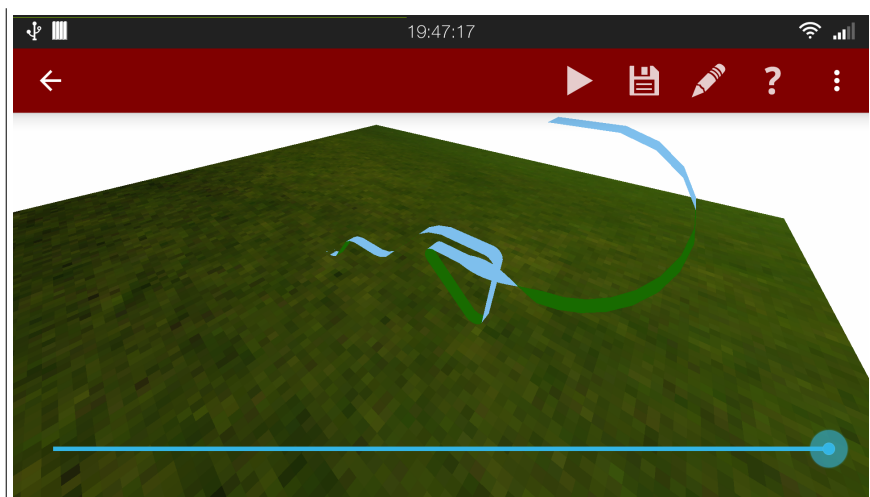
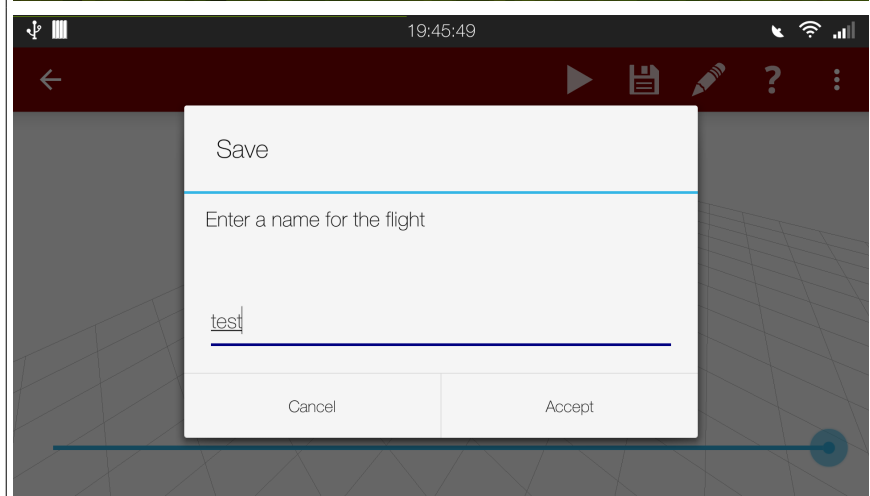
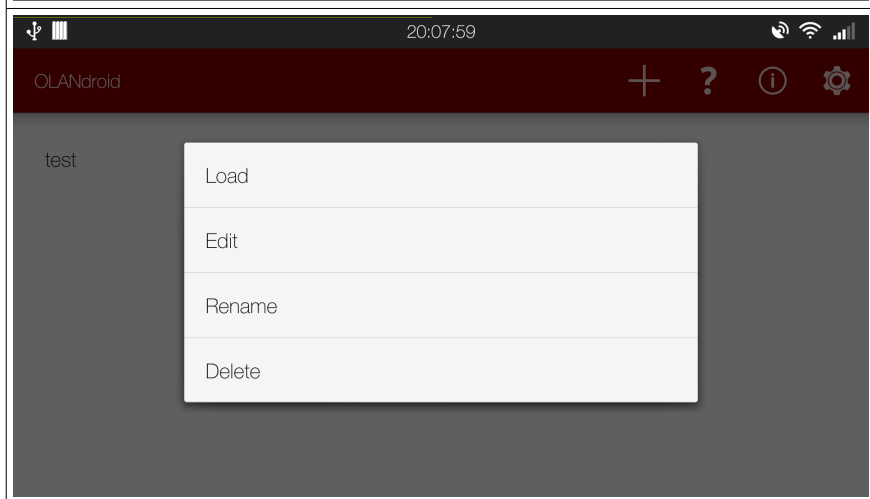
The about activity –  
“AboutActivity” – detailing  
some information about the  
project



An example of a fairly  
complex flight in  
“VisualisationActivity”



The same scene as above,  
but with a different colour  
theme and half animated

	<p>A different flight in the “VisualisationActivity”, with a different ground style – the textured one</p>
	<p>Name entering dialog available in both the “VisualisationActivity” and “FlightManagerActivity”, letting a user enter a new name for a flight</p>
	<p>Context menu brought up by long-pressing a flight in “FlightManagerActivity”, giving a few choices for operations on a flight</p>

## 9.6 Document history

Version	Date	Change
0.1	2015-04-10	Document created
0.2	2015-04-28	Draft released
1.0	2015-05-04	Final version released

