

**A WebGL-based aerobatic visualiser using the
OLAN(One letter aerobatic notation) catalogue
to provide users with a means of generating
interactive 3D representations of manoeuvres.**

Final Report for CS39440 Major Project

Author: Craig Heptinstall (crh13@aber.ac.uk)

Supervisor: Prof. Neal Snooke (nns@aber.ac.uk)

29th March 2015

Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a MEng degree in
Software Engineering (G601)

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

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

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

Date

Acknowledgements

I am grateful to...

I'd like to thank...

Abstract & Background

The aim of this project is to create a 3D representation of aerobatic manoeuvres, primarily using the OLAN notation(or one letter aerobatic notation). Both real-scale and remote control aerobatic planes and helicopters use the notation to describe a set of manoeuvres in an overall routine, usually including translating the notations into Aresti symbols. The Aresti symbols came before OLAN, but OLAN was developed to make it possible for pilots to write down quickly their planned routines.

The primary element of this project will involve allowing users to insert their notated routines into the application(via an input box on a web-page) thus producing first the ribbon shapes of the manoeuvres, followed by the ability to see a craft fly the route. Although there is only a finite amount of manoeuvres possible from the Aresti catalogue, each OLAN notation can have its own parameters. This can range from entry length into a loop or turn, to the number of rolls in a section of a manoeuvre. The application will also need to be taking into account flight speeds, and possibly wind and gravitational effects. The life cycle will be chosen after the initial list of requirements is apparent and in an organised manor.

WebGL will be the language used to create this application, with hopefully object-orientated Javascript to power the application. Libraries helping towards the graphical/ visual side of the application may be required to give greater flexibility and better aesthetic value. All considerations will be found in the analysis and design stages of this report.

CONTENTS

1	Background & Objectives	1
1.1	Background	1
1.2	Analysis	3
1.2.1	OLAN and Aresti interpretations	3
1.2.2	Application functionality interpretations	5
1.3	Process	7
2	Design	8
2.1	Overall Architecture	8
2.2	Back-end logic and design	9
2.2.1	Design patterns	9
2.2.2	Module diagram	10
2.2.3	Object structure and storage formatting	13
2.2.4	Naming conventions	15
2.3	User Interface	15
2.3.1	GUI Use-case	16
2.3.2	CSS and wire-frame design	18
2.4	Planned use of services	20
3	Implementation & Testing	22
3.1	FDD project approach	22
3.1.1	Iterative Implementation	23
3.2	Final status and progress	35
3.2.1	After-test fixes and developments	37
3.3	User testing	37
3.4	Implementation and testing review	38
4	Evaluation	41
4.1	Anlaysis and design decisions	41
4.2	Application and testing evaluation	41
4.3	Project improvements	41
4.4	Self-Evaluation	41
	Appendices	42
A	Background research and analysis	43
1.1	Initial project Gantt chart	43
1.2	Inital requirements analysis	45
1.3	OLAN understandings	47
1.4	Revised FDD Gantt chart	53
B	Third-Party Code and Libraries	55
C	Code samples	56
3.1	Off-canvas HTML code, showing how menus are hidden	56
3.2	JSON OLAN instructions example	57

3.3	RequireJS setup code	57
3.4	Converting JSON representations into vectors	58
3.5	Using local storage to export scenarios	58
D	Application tests	59
4.1	Usability testing for the GUI front-end	59
4.2	Test Unit code for converting JSON to manoeuvre instruction object	60
4.3	Test Unit code for checking lights and ground are created successfully	60
4.4	Usability testing for cameras	60
4.5	Test Unit code for animation controller getters and setters	61
4.6	Test Unit code for checking import and export to JSON/ local storage functionality	61
4.7	Usability testing for GUI options	61
4.8	Usability testing providing parameters between OLAN	61
A	Annotated Bibliography	64

LIST OF FIGURES

1.1	Image of my initial user stories after my first meeting, short and concise requirements.	2
2.1	Model View Controller diagram for my proposed application. Shows the planned interactions between different subjects within my application.	9
2.2	Module diagram displaying the various communications between modules, and how they are connected to the GUI in the MVC way.	12
2.3	Use-case diagram detailing user options, the canvas and OLAN input.	16
2.4	Flow chart showing how the application should run in accordance to loading up and then allowing the user to begin using various functions.	18
2.5	Wire-frame design detailing desktop view of the application, including the open menu for the 'off canvas feature'.	19
2.6	Wire-frame design detailing mobile view of application with navigation bar in minified version.	19
3.1	Screenshot of GUI created with Foundation, JQquery and HTML markup.	24
3.2	Model of plane loaded up using JSON and three.js object loader. This plane will represent the onboard camera, and fill be the object that will be assigned locations along an OLAN path to show flying the flight plans entered by users.	28
3.3	Figure illustrating that where the vector is turning right to complete the diagonal line up, it is becoming flat with the X axis rather than curving to the correct angle.	29
3.4	Movie reel overlaying the bottom of the canvas, with some figures loaded after the user typed in a string of OLAN.	34
3.5	Burndown chart of features implemented and tested fully. X axis represents number of features to implement, whilst Y axis shows the date that the features have been implemented by.	36
3.6	Feature progress reports for each of the feature sets implemted up to this point in time. The predicted month of completion is shown at the bottom, and tasks where they are not complete and tested fully are ones without a green background.	37
3.7	Framerate counter shown in bottom right of the application.	39
3.8	Average rate per device(Graphics RAM in brackets)	39
3.9	Average frame rate per browser (all latest versions of browser on 04/05/2015)	39
A.1	Initial gantt chart, outline times throughout project activities, to be updated as time passes during the project span.	44
A.2	Requirements brief created before a meeting to clarify questions on the initial requirements. The pdf continues onto the next page.	45
A.3	A document created before a second meeting, outlining my current undertsanding of the OLAN language, and how manoeuvres can be constructed from smaller single-element ones.	48
A.4	More detailed Gantt chart, focuses on the implmentation and testing stratedgy of each feature. As you can see, prioritised tasks are the first to be completed.	54
C.1	HTML code showing the page is divided into two, one for the canvas, and one for the menu. The 'off-canvas-list' has sub menus that allow more menus to be hidden to the left.	56
C.2	JSON code showing a vertical upline defined by JSON instructions.	57

C.3	Snippet of code showing how RequireJS starts the application. The RequireJS library is loaded onto the page, the configuration telling the library the base of the application JavaScript files is set, and then is told to start by running the main.js file.	57
C.4	58
C.5	58
D.1	60
D.2	60
D.3	61
D.4	61

LIST OF TABLES

1.1	A table of prioritised features and tasks that I would like to have implemented by the end of the project. The lower ranked items will only be started on completion of higher tasks.	6
3.1	Collected statuses of all features required, with their duration and percentage complete. Note: the dates shown the task was completed working on also include time done testing, and writing this stage of the report.	36
D.1	Usability test table for testing the GUI.	59
D.2	60
D.3	62
D.4	63

Chapter 1

Background & Objectives

Before commencing the design of the application and the project planning, it is important to have analysed what I hope to have achieved at the end of my project time, and also what steps I will need to be taking to implement each feature. As I will mention later, choosing the best fitting life cycle methodology will play a big part of how I shape my project and create each feature whether it be by priority, size or difficulty. This section details my understanding for my project requirements, steps I am going to need to take, and as I would prefer my project to be similar to and FDD one; developing an overall model and building the list of requirements and features.

1.1 Background

The choice of undertaking a project such as this one was due to two combining factors: maths and an interest to learn graphical programming. The fact that this application will require me to learn graphics, and how to implement visual effects representing the requirements of the project in ways completely new to myself. As graphics is something I have not had much to do with in the past, this project appears both exciting and daunting task due to the learning curve I will need to take. As for the maths factor, I can assume quite a lot of maths will be involved (especially for creating curves, rolls and turns along most of the manoeuvres) which I enjoy learning about.

In terms of the history of the topic, OLAN was originally developed by Michael Gorden in 2006 [1] and was designed to provide shorthand notation for pilots planning out aerobatic routines without having to draw out the full Aresti diagrams. In recent years, the OLAN notation became used much more until because of licensing issues with the original owner was taken off-line. Because of this, a new form of the notation has been created in a more open source way paving the way for applications such as this project's intended aim. Although in this report and my planned application itself will be still referring the notation as OLAN, the new re-make of the language is known as the 'OpenAero language' [2]. This is based off of the original, yet is open and allows anyone to use it. In combination with this, the creators of the OpenAero language also developed a web-based application [3] that allows the conversion of the notations to 2D Aresti diagrams. This is somewhat similar to what I hope to achieve, but alongside plenty more features most importantly the ability to see a plane perform the moves.

As for Aresti, named after its conceiver Jos Luis Aresti Aguirre [4] is the diagram format that OLAN achieves, and represent informative diagrams showing the shape of the routine, direction

of travel, rolls and sharpness of turns. Aresti diagrams also can include angles or turns, ranging from 90 degrees to 270. Each diagram usually has a name [5], relating normally to the shape of the manoeuvre, though some are more commonly known to pilots rather than the regular user. The OLAN notation for each diagram usually attempts to try describe the manoeuvre with the letter used, such as 'o' for a loop, or 'z' for a shark tooth. The full list of manoeuvres, including their OLAN notation and full name can be found on the OpenAero [2] site.

Upon starting this project, several meetings with the project supervisor are planned each providing more detail of the initial requirements. The project plan considering the requirements has been made into a rough Gantt chart which can be found in section ?? of appendix A showing my plan following. Because I have already attended several meetings at this stage, I can provide a fairly accurate time-scale to work with.

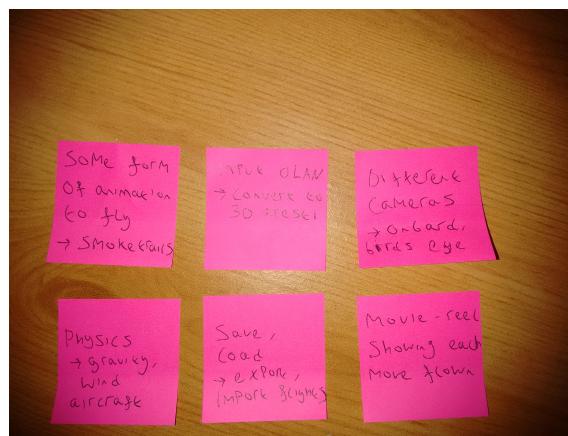


Figure 1.1: Image of my initial user stories after my first meeting, short and concise requirements.

The initial required application can be broken down into an extensive(but less detailed) list of main functional requirements. These are as follows:

1. Provide a web-implemented tool that allows input of the OLAN 1 None-IE due to WebGL capabilities. Will it use a simple JSON file to store notations? characters as a string format, alongside possible click functionality.
2. Relate each notation or set of notations to a certain procedural movement(rotations, movements etc.).
 - Must consider parameters in some of the notations, such as the speed of entry.
3. Provide a means of linking up these movements in such a way into moves, or the angle of the plane. They should produce a fluid manoeuvre.
4. Display this using WebGL. Libraries to consider that could help. Begin by initially testing simple shapes to move and fly around, then add textures, and plane structure.
5. Are libraries OK to use? with some of the movements:
 - glMatrix- JavaScript library for helping with performing actions to matrices [6]
 - ThreeJS- Another JavaScript library, good with handling cameras and different views [7]

6. Allow user to add different effects such as wind, gravity changes and other physics. Could be better to implement these last, as it will be easier to test pure functionality of rolls etc. first, then figure out natural physics.
7. Add functionalities of different viewpoints(on-board views, side views) to application.
8. Possibility to add function to save (using local storage?) users different sets of manoeuvres?

The list shown above also has an accompanying report in section ?? of appendix A which I created after my first project meeting. This report includes the list of initial requirements, alongside footnotes, and also a detailing of the methodology and process I plan to follow. The document can be found in the appendices section of this report.

In addition to the list, I feel it should be highlighted why and what language I will be using to create my application. I chose WebGL over OpenGL because of two reasons, the first being that I like the idea of being able to run an application such as this simply in a web browser, without the need for any compilers or platforms installed on the user's device. The other reason being that I already have good experience using JavaScript, and this will help when it comes to the writing code, making sure the coding style is appropriate, and maximising any features it could bring to my application.

1.2 Analysis

Before I begin using my chosen life cycle model, It is important I analyse the overall model and requirements of the intended application. I plan to analyse two items: the requirements analysed by time, effort and difficulty, and also a breakdown of the OLAN and Aresti manoeuvres. The second of which I will break down to their primitive forms, hopefully finding out how I can make my application create each manoeuvre as simple and efficiently as possible.

1.2.1 OLAN and Aresti interpretations

The best place to begin with my analysis is to look in more detail at the OLAN manoeuvres individually, by breaking down each manoeuvre into their primitive elements. As with the previous section, alongside I have attached a document to the appendices of this report, detailing the main and most important manoeuvres I believe are key to finding primitive shapes. I began this by firstly organising each OLAN and their corresponding Aresti shapes into sub groups.

Of these, there were:

- Single element- These include manoeuvres that can be described as one fluid movement. For instance, the OLAN letter 'd' would mean diagonal line up, which requires only one action to complete the manoeuvre.
- Two-element- This group includes any manoeuvres that require two separate manoeuvres to complete a given manoeuvre. An example of this could be the 'z' notation, also known as a shark tooth. This shape requires both a diagonal line up followed by a vertical line down.
- Loops- These like the single element moves, consist of a single manoeuvre, and can be combined to make other manoeuvres.

- Loop-line combinations- These are loops that are a combination of a loop and a single or two-element manoeuvre.
- Double-loops- As the name states, these are manoeuvres that contain two loops.
- Humpty-Dumplings, Hammerheads and Tailslides- Each of these manoeuvres represent specific shapes, such as a 'humpty dumpling' which consists of a bump shape comprised of a 180 degree turn to come back down vertically. As the previous, these shapes can simply be combined from single element pieces.
- Complex 3 rolling elements- The naming behind this group of manoeuvres comes from the fact that each contain a set of 3 elements to create the entire figure.
- Special and 'oddball' figures- The group of manoeuvres that are more complex in such a way that they require special sections not available from any of the other groups. One example of this is the OLAN letter 'f' which represents a flick. This comprises of rolling the aircraft 360 degrees along its horizontal line.

Another important part of the OLAN analysis that I had to understand before proceeding was the possible parameters, prefixes and postfixes that can be attached to manoeuvres.

Looking at prefixes first, each one-letter-notation, **some moves** are able to be reversed, or inverted. These are so:

- r - Reverse, meaning to order of how each part of the manoeuvre is done. For example, placing the letter 'r' before 'c', would represent a Cuban loop flown in the opposite order.
- i - Inverse, meaning that each part of the manoeuvre is done in the same order, but inverted in terms of value. For example, the letter 'i' before 'c' would mean that rather than looping upwards, the loop would go down. A diagonal line upwards would become a diagonal line downwards.
- ir - This is simply a mixture of both the previous. The manoeuvre fixed to the end of this postfix would both be inverted and then reversed.

In addition, there are a number of postfixes that can be used with some manoeuvres particularly with roll or turn based figures. For example, manoeuvres containing rolls can be represented with the prefix angle of turn in multiples of 90 degrees, and a postfix of the amount of rolls along the same part of the path. So in one example, the notation '2j2' would represent a 180 degree turn, whilst rolling the aircraft twice. Alike the prefixes for inverting and reversing manoeuvres, the parameters are not available on all the manoeuvres in the OLAN catalogue. Again, the full list can be found on the the OpenArea site, or see the my OLAN understandings in section ?? of appendix A.

One final set of optional parameters that could be required of my application to handle are the positions of manoeuvres. Although not strictly in part of the OLAN catalogue, it is already available in the OpenAero application. These parameters are structured (x,y) with x representing the amount of horizontal distance and y the vertical distance from the end of the previous manoeuvre to the start of the current manoeuvre. These can also be negative values to ensure the user can control the position fully.

The main reason for the need of this is that simply all manoeuvres cannot fully follow each other straight after each other. In real-life if a manoeuvre made the pilot finish near the ground, and the next move required them to perform a diagonal line down, they would hit the ground. Obviously my application will attempt some form of validation and checks, but offering the option to the user is a very useful feature.

From my analysis in the groups listed above, a number of assumptions can be made.

1. I have already deduced that all non-single manoeuvres that do not include loops or rolls should be possible to be made from a set of single element manoeuvres.
2. In total, any manoeuvre can be constructed from one of three primary moves: diagonal and straight lines, curved lines, and turns and rolls. Each of which should be able to carry parameters.
3. Each curve should be possible to be created based on 45 degree increments, as this is the smallest change of angle in any manoeuvre, and all other angles seem to be in multiples of this number. This will shorten the need for multiple commands for different ranges of angles when programming in the manoeuvres. Because the changes in angle will need to be by a curve, interpolation will be required to make the change in angle smooth and realistic.
4. Turns, curves and rolls will all need parameters, as some curves are steeper and shorter than others. The same goes for rolls, where you can choose anything from quarter to 3 rolls, and in turns the angle of turn should be specifiable.

By considering my analysis of the set of manoeuvres, I can now envision what manoeuvres will be possible to create easiest, and prioritise my work better. The next section of this report will group up the functionality of the application with my OLAN manoeuvre findings.

1.2.2 Application functionality interpretations

Upon having my initial meetings with the project supervisor, and on analysis of the OLAN catalogue, I can now make more final judgements on what I want to have been achieved by the end of this project. As I discussed in the background section, I had already created a set of initial assumed requirements. Though, some of the features raised questions, such as the possibilities to use libraries for the physics and graphical side. After some research I can now define a more cohesive and stable list of features and requirements of my application. Rather than in a list format though, here I will group features together and discuss the most important factors.

Starting with the underlying functionality based on my OLAN interpretations, what I hope to be achievable by the end of this project is the possibility to allow users to enter any length of space separated OLAN characters into an input box, alongside any possible prefixes, postfixes and parameters. I hope to achieve this by making my system as general as possible, and by using very abstract methods that can support any given move input. One way I could think of implementing this now would be to break each manoeuvre into a set of instructions, each one saying which direction to move, the angle and length. By breaking each manoeuvre into these, I could use a single method to construct each manoeuvre on the fly. This is important as it would then allow for the user to play through the manoeuvre in an animated fashion and see the aircraft move.

Table 1.1: A table of prioritised features and tasks that I would like to have implemented by the end of the project. The lower ranked items will only be started on completion of higher tasks.

Rank	Description
1	Support broken down manoeuvres, via JSON, convert to vectors and ultimately figures
2	Allow for each manoeuvre to link together, start at end of previous
3	Animate along a path of the manoeuvres
4	Create cameras, onboard and birds eye
5	Ground and lighting additions
6	Saving and loading of inputs
7	Manoeuvre validation
8	Physics options, different aircrafts
9	Movie reel functionality
10	Mobile compatibility and nice GUI

More generally, one of the questions asked by myself at the start of this project was if libraries could be used to help create the graphics and physics. By now, I have had more meetings with my supervisor and found this was allowable. The main library I have considered after this has been the ThreeJS [7] library which acts as a good wrapper for controlling a wide range of objects in a scene. This library will allow me to easily manipulate vectors, cameras and lighting which will form the basis of my application. I will discuss the use of this library when it comes to planning each of the features in my design.

This brings me to the scene and cameras that I hope to have working to a good standard. For the scene, things such as the ground terrain are not so important, and can simply represent a flat land, while lighting could be added, but perhaps after fundamental features are added. As for cameras, I would like it that there is a set of two cameras: one for navigating and viewing flight paths at different locations and angles, and another camera which would be on-board, like a nose-cam.

The save/ loading of flight paths is a lower ranked task, but something I will definitely be planning to have implemented by the end of the project. Having looked into various methods of saving the OLAN entries, the best way I have found is to use a combination. One which would use local storage, and one that would export to JSON. The first of which I found out here [8] is particularly useful as my project is web based.

There are four features I would also like to add, but I will make these optional for now, and place these after the previously mentioned features. The first of which would be to validate the manoeuvre entry. Currently in the OpenAero application there is a check that looks how close and where manoeuvres are placed on the canvas, and for my application it would be ideal to have a check that looks if manoeuvres are actually possible from the current rotation or placement of the last manoeuvres ending position. Another big check would be if the current path of the aircraft was to hit the floor, a check should be made.

Physics is also an option I would like to include, such as wind, type of aircraft(each aircraft could be more difficult to navigate corners, meaning wider curves). And relating back to the function of playing the animation once the manoeuvres are drawn, an idea passed onto me from my supervisor was of a 'movie-reel' function that would show a mini image of the current manoeuvre being played, showing the animation progressing through each one. Again, this is another feature I would prioritise less, and implement after other key features.

Finally, a more smaller task I think I should set myself is to make the application mobile compatible. As most phones also now have WebGL capabilities, making the application run on mobiles should be possible. This is more a GUI centred feature on the site itself though, and should be added nearer the end of the project.

1.3 Process

Moving onto more project and time management specific items, the process in which I follow can have a large effect on what features I complete on time. At this stage, I would suggest using a hybrid of both the waterfall methodology alongside feature-driven development to create my application. The reasoning for this starts with the way I have already created a list of features in my analysis, which is already part of the FDD life cycle. This would mean following this, I would be able to simply iterate over each feature, plan, implement and test each one by one. This is an ideal trait that comes from using this methodology which allows me to create each feature separately, and more importantly by priority. Then for instance if I was not to finish the entire list I outlined, my application would still have a good deal of functionality on offer. A guide I found on the agile modelling site [9] explained this to me well. If I were to use the waterfall method alone, it might mean I try implement too many items at once, yet not finish certain parts that rely on others. This would result in an incomplete and less functional program. I feel like there is also a clear part of the scrum methodology put to use here, as I showed with some user stories, and the prioritised list of features.

The second reason, and reason for including the waterfall cycle in my hybrid approach is because I would like to create a more big up front design before beginning implementation. This is where my approach is going to be different from a solely FDD way, where I would usually have to design each feature before implementation and testing. I prefer the way of knowing how the entirety of the structure of code should look before I begin, yet keep the structure as loosely connected as possible to ensure that features avoid relying on each other to an extent where if one is broken, the rest is broken.

Because I have chosen my methodology in this fashion, has meant I have been able to create a Gantt chart based on stages of planning and design, and also what features I hope to have accomplished at times throughout my project. This, coupled with a work blog, will allow me to track my progress throughout, and compromise when time is needed, or move along the list of features if time is available. I will update my Gantt chart as time progresses in relation to my blog, where I will then be able to see where progress is up to overall.

As for implementation and testing, I will carry out these as normally in the FDD way by implementing and testing each feature one at a time. In the implementation stage of each, I will ensure that decoupling is a priority meaning any developing code does not damage any current working functionality of another feature. The implementation and testing schedules of each feature can be seen on the updated Gantt chart in section ?? of appendix A. This time, the Gantt chart tasks in to consideration the difficulty and time required to implement and test each feature.

Chapter 2

Design

On completion of my analysis and background planning for the project, I can now look at more detailed design for the application overview, and the individual component it comprises of. Because my application will present both a front-end GUI and back-end JavaScript and WebGL, I will split my design into two sections. The reasoning for this is because I would prefer to have the GUI and logic of the application to be decoupled, so that the code can be changed in the JavaScript easily without having to affect the user interface. Other than planning the application itself, it is also important I plan what key assisting services I use, such as how I intend on Versioning my code, to how I will deploy my code.

2.1 Overall Architecture

As a summary of the architecture of my planned application, the best way to design in more detail is to lay out a primary design pattern. For the purpose of an application which will allow for user interaction, which will be processed by code in the back-end, I have decided to go ahead with a Model-view-controller approach. Using the MVC pattern means I can separate the GUI from the logic code as I wanted to, and have the GUI exists without knowledge of the back-end. This is also the same with the model, where the primary code for calculating manoeuvre movements and animations should be possible without knowledge of the view, but instead use the controller as a intermediary. See figure 2.1 for the MVC pattern I plan to use.

Because I have already chosen three.js as library of choice for creating any graphics and physics objects, these will already form my model or models. Each manoeuvre for instance will be reflected as a set of vectors which will form the shapes of the Aresti flight paths, and should be accessible and changeable through the controller to the view(in this case the canvas in my GUI).

The controller will be the most crucial part I implement, as it will need to be able to communicate with the three.js objects, the canvas displaying the animation, and detect controls from the user. For this piece of the pattern, I will enforce some other design patterns to

Finally, the view will be represented as the canvas and controls on the web page. Since on both the canvas and the options menu can have an affect on the model, the controller will listen for changes on either, and then call the relevant operations to affect the model, and again reflect this back onto the view. The view will not know anything of the controller nor the model, so adding any new options or displayable information will be easier and not conflict with any current elements.

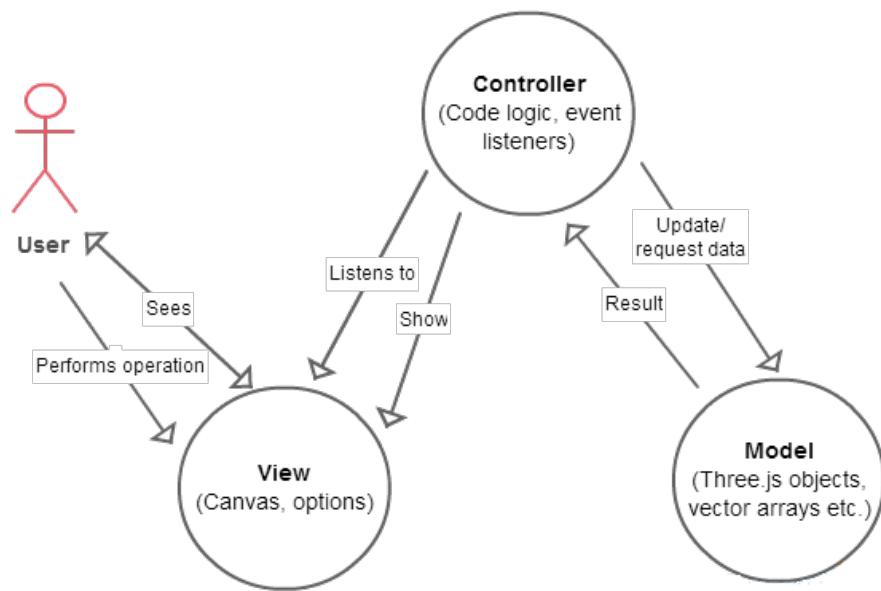


Figure 2.1: Model View Controller diagram for my proposed application. Shows the planned interactions between different subjects within my application.

2.2 Back-end logic and design

The first of the two design categories concentrates on the JavaScript that will act as the functions to run each of the proposed features of the application. The code on this side will be responsible for maintaining contact with the GUI, and more importantly the WebGL canvas on the web page. To make the code be as maintainable and run effectively as possible, I will look into a variety of design patterns.

2.2.1 Design patterns

Because I am using a hybrid of waterfall and FDD at this stage of the project, using a changing range of design patterns for my JavaScript and GUI architecture is possible. Through the implementation stage of this report, some of these may patterns mentioned may not be used anymore and replaced for different ones depending on changing requirements and progress. Using the implement, refactor and test iteration approach should allow for this.

The first design pattern that would be good useful relating to the controller is known as an observer pattern. The observer pattern means to listen on an event or events in an application, and then call the relevant action. The JQuery library, which I plan to use throughout any GUI related tasks will be of a great use here. This library will allow me to add simple listeners on elements on the web page, and set methods to be called on any click, hover or other events. In terms of how I will place this pattern in my architecture, a standalone file will be responsible for listening to any options or menu changes, whilst another JavaScript file will be responsible for listening to the canvas events such as rotation or zoom. One web article here [10] supports the advantage that the observer pattern will help towards decoupling.

The next design pattern I plan on implementing into my application relates again to the MVC

architecture I will be using. The builder pattern will allow me to append and edit any HTML on the page(such as options, check-boxes, loading content back into the OLAN input box) easily with data retrieved from the model. JQuery again should help to provide a means of changing content, because of more built-in method it comes with. There should be a handler class in my application specifically for controlling the page content.

A third design pattern which will be important when it comes to loading up the application will ensure that all the data necessary to run the application is ready before the user can perform any actions. Known as the Lazy initialisation pattern, this is a style of coding that means whenever files or data is being loaded, the rest of the application should either wait, or prevent other relying features from being initialized. One motivation for the need for this pattern in my program is because of the vast amount of manoeuvre data, and model data for terrain or for aircrafts means that functionality such as animating and drawing flight paths on the canvas may be already ready for use from the user before all the data is ready. In this case, it could cause errors and even crash the application. Therefore, making sure that the application does not progress loading before data is ready is of great importance. Martin Fowler mentions this pattern in his site [11] and whilst says this pattern should be avoided if possible due to responsiveness being hindered, he agrees that this could be used to solve performance problems.

Although the previous design patterns have been chosen, others were considered when planning the application but were not suitable or better options were available. One such pattern, the Mixin pattern allows JavaScript functions to be inherited from other classes or inner functions. For instance, these would be useful as a means of decreasing repetition of functions. In my proposed application, I thought about the possibility of using a Mixin style architecture to allow functions that would draw both the manouevres on the canvas, and the manouevres on the movie-reel. The reason I have decided not to use this pattern falls to the issue that by making an object extend and hold code from elsewhere could make it harder to maintain, see where the function comes from, and uncertainty of location of any bugs I may come across while developing. While researching other possible design patterns, a book written by Addy Osmani [12] was especially useful as most known JavaScript design patterns were explained, and their advantages and disadvantages pointed out.

2.2.2 Module diagram

Because much of the code I will be implementing will hold various Three.Js objects, I have decided that modulating methods and objects will be better than simulating classes seeing as JavaScript is a class-less language. This is a pattern known as the module pattern. Although this may appear less object orientated, modules allow for more robust architecture where units of code can be separated and organised. Modules are slightly similar to classes in the way they can hide code that should not be accessible to other modules by encapsulates privacy. When code is modulated, and then that module is called upon by another, only a public API is returned, and other methods in that module are kept private from being used in other parts of the application. These private methods are good for use as supporting methods, holding such things as calculations, or private variables for getters and setters. Again, this is a similar case to the traditional class diagram.

There are currently a selection of libraries that allow for modules in JavaScript. The most prominent, and the one I would like to use is called RequireJS, which promises the increase in speed and quality of code. RequireJS works by dynamically loading JavaScript files on the fly, where the code has from the other module is usable once loaded into the module calling it. Once modules

are loaded into an object form in whatever the developer needs to name it, its public variables and methods can then be accessed. See figure 2.2.2 on how modules are used. In my case, modules would be useful in enforcing the MVC pattern in the way that it will help towards hiding code between the view and the model.

```

1 require(["helper/util"], function(util) {
2
3     // This function can not be called by another module
4     function private_function() {
5         util.Method() // Can call public methods in the util file
6     }
7     return {
8         public_function: function() {
9             // can be called if this module is loaded into another
10        }
11    }
12 });

```

Listing 2.1: Example showing how RequireJS loads in another module or JavaScript file which in this case is loading up the util JavaScript module and naming it as object 'util' for use in the code

In order to create a basis for modulating code, I should first look to separate the features I listed in the analysis section of this report into categories. These categories will then help me to determine how I could structure my application in as best object orientated way as possible.

The categories I have been able to come to are:

- Main- initiating other modules, beginning the application.
- Animation- Playing, and controlling speed, physics of animation.
- Loading manoeuvres at start of application.
- Saving and loading animations
- Cameras- Creating and controlling cameras movements
- GUI controlling- control and edit GUI controls, and appearance from back-end. Also including the possible movie reel live animation.
- Canvas controls- Allowing the user to move along the canvas, and zoom.

Now I have a stable list of categorised features, I am able to create a diagram shown in figure 2.2 to represent what modulated layout my application will use. Because of the way modules handle public and private variables, have public and have private methods, means that this is very reminiscent of a standard class diagram.

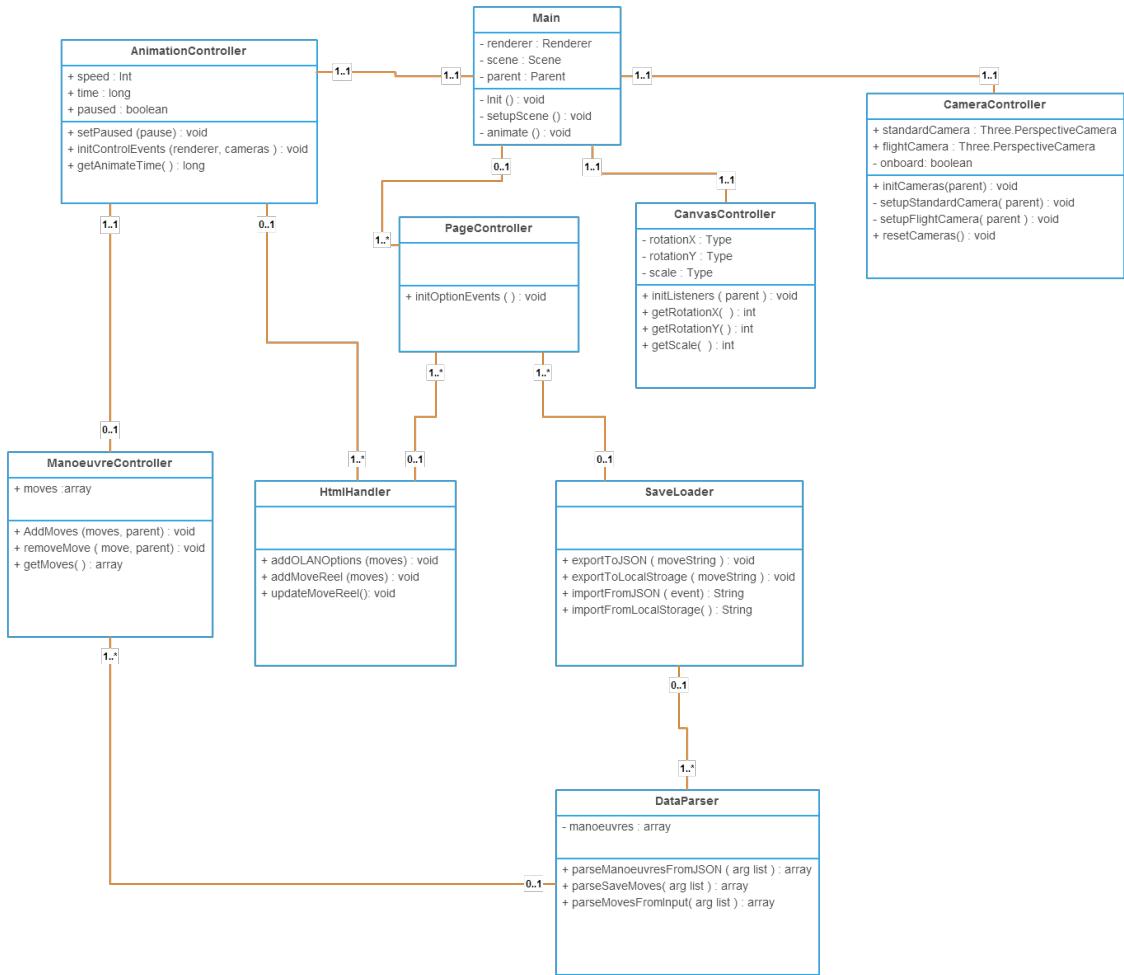


Figure 2.2: Module diagram displaying the various communications between modules, and how they are connected to the GUI in the MVC way.

This module diagram shows what will be the communications between modules in the application, providing insight into what each module should contain in terms of important variables and methods. Ensuring that only modules that require certain pieces of information from another module has sole access is important to reinforce the module pattern mentioned earlier. It should be noted here that although the diagram shows the main modules that will oversee most of the features of the application, when implementation occurs later on in the project, more modules or JavaScript files may be added to support or lighten the load of functions. This is supported by the good practice of **keeping files and functions short and concise** for easier maintenance and readability. The following descriptions have been kept brief, and key connections within the diagram discussed.

The first module proposed from the diagram will be the module entitled 'Main', which will be responsible for setting up the application, and hold onto key variables such as the renderer and scene objects to allow for anything to be drawn and animated on the canvas. This module will be called from the HTML and then will begin any necessary calls to set up the listeners for the cameras, canvas and options. The most important call will be the call to set up the animation controller module.

Following on, the animation controller module will be one of the largest and most important parts

of the overall architecture, as it will be responsible for both animating flight paths, loading up the OLAN manoeuvres at start of the application, and relating any actions from the user to do with OLAN selections or playing and pausing, as well as changing animation speeds.

The animation controller module will be in communication with the manoeuvre controller, which will be a logic heavy class because of the calculations it will require to compute and construct the sets of vectors of the Aresti shapes representing each OLAN notation. As the diagram shows, the application will first get the user OLAN input from the web page via the HTML handler module, then search via the dataParse module and create the manoeuvres array object within the manoeuvre controller holding all the calculated manoeuvres, and return this in a 'get' method to the animation module for placing on the canvas.

As mentioned earlier, the HTML handler module will be the first means of contact to the user of the application, retrieving input from the OLAN box, getting values of any check-boxes, and also setting any values. This module will be using the JQuery library as well as the Handlebars library as the primary way to communicate with the front-end, by directly referencing ID's of divs on the web page.

Two other controllers that should be mentioned are the camera and canvas modules. Both modules will be responsible for setting up their elements when the application starts up (including the different sets of cameras, locations, lighting and ground effects) and for listening to events on both of their respective related front-end sections. Whilst the canvas module will only need to listen for events on the canvas and then reflect this by modifying the canvas directly, the camera controller will need to communicate with the manoeuvre module if the user is using the on-board view to get the current location on the flight path and then place the camera there.

The final module to be highlighted in the diagram is the save and loading module, responsible for the storage of user input and flight paths. Because I suggested two means of saving flight paths which were JSON and Local storage, there is the option to create a module for both of these, one for handling saving data to files, and one for in the browser. At this stage, keeping all the save and load logic in one module will suffice, as there should not be many methods required, so files would be relatively short. The module will then be able to share methods between both means of saving for preparing data. There is a trade-off currently between saving OLAN as the rendered set of vectors, or simply saving user input. The first would mean the application would be ready as soon as a flight is loaded, but would take more space to save, whilst the other way would mean slower loading but much shorter save data. I will consider both methods when implementation occurs.

2.2.3 Object structure and storage formatting

It is imperative that the structure of data, especially the OLAN instructions for construction of each manoeuvre is organised efficiently and be as accessible as possible, to consider the speed of the application running in a user's browser. As it is planned to represent each broken down manoeuvre into a JSON string of instructions, they should be easy to read, maintain and add to. The proposed structure of the file holding the JSON will be as follows:

```

1 {
2     "catalogue": {
3         "manoeuvre": [
4             "variant": [
5                 "component": []

```

```

6           "_pitch": "NIL",
7           "_roll": "NIL",
8           "_yaw": "NIL",
9           "_length": "1"
10      }, {
11          "_pitch": "POS",
12          "_roll": "NIL",
13          "_yaw": "NIL",
14          "_length": "1"
15      ],
16      "_olanPrefix": "",
17      "_name": "example cuvre up"
18  },
19  "_olan": "u"
20 }
21 ]
22 }
```

Listing 2.2: A JSON means of holding break downs of manouevres with each one holding information on different variants of the move such as inverse and reverse and description of the OLAN notation

The JSON example shown in listing 2.2.3 displays the planned layout of my manoeuvre breakdown. The first element labeled 'catalogue' will hold the entire list of OLAN notations (without the prefixes and postfixes for reversing or inverting moves), with each holding another array of variants. Within each variant, the data acting as instructions for the how the application draws the paths is contained. Each 'component' will hold four key pieces of data:

1. Pitch - Instruct to direct the manoeuvre to fly upwards or downwards on the Y axis
2. Roll - To roll to the left or right on the aircraft's Z axis
3. Pitch - To turn left or right on the X axis
4. Length - length of the manoeuvre

By using these four attributes, the manoeuvre module will be able to go one by one through each creating a new vector based on the previous vector with these effects added. As you will see in listing 2.2.3, the value of each will either be Nil, positive or negative. This makes it simple to tell the application if the manoeuvre is for example banking up, down, or remaining on a straight flight. The length attribute will be used to tell the application how far to move along the paths current Z axis after performing a move. If pitch, roll and yaw are all nil, the plane will follow a straight line. The length attribute will be very useful for adjusting the amount a pitch up spans; if it is small, the curve upwards will be tighter, and a larger length will mean a longer less noticeable curve. This will be especially useful for some requirements outlined in the analysis and feature list of the project concerning different aircraft types carrying different traits (some aircrafts may require more distance to bank to the same angle as others).

As mentioned in section 2.2.2, there are two options for the data structure of the saved data from flight plans. The first option which discussed storing the pre-compile vectors from OLAN entry before saving could appear as such:

```

1 {
2   "Manouuvres": [ [
3     "Move": {
4       "Vectors": [ [
5         {
6           "Vector": "2, 1, 0"
7           "Rotation": "0"
8         },
9         {
10           "Vector": "5, 1, 0"
11           "Rotation": "0"
12         }
13       ], {
14         "OLAN": "u"
15       }
16     ]
17   }

```

Listing 2.3: A JSON means of holding break downs of manouevres with each one holding information on different variants of the move such as inverse and reverse and description of the OLAN notation

While the second option which will be the one I will be looking to implement first with respect to time could be simply:

```

1 {
2   "OLAN": "o id b"
3 }

```

Listing 2.4: A JSON means of holding break downs of manouevres with each one holding information on different variants of the move such as inverse and reverse and description of the OLAN notation

Once other proceeding features have been created and time is available to add the save/load feature, then both options will be considered again. For the time being, it would be easier to choose the second option because functions created to build flight paths will already be there so they could be re-used to process the OLAN input again.

2.2.4 Naming conventions

The final considerations that need to be looked at in the back-end are naming conventions. It is important that variables, methods and modules are named accordingly to their function, and so that they are easily readable. For the purpose of my application, I will be ensuring to use the Google recommendations for naming conventions in JavaScript. The guide, which can be found on their style-guide site [13], details variables, global variables, functions and class names that should be used when coding any system. The naming conventions in my application should all follow the same style throughout.

2.3 User Interface

Moving onto the GUI design, which will provide the users a means of adding OLAN input and animating the simulator must also be designed taking into account the placement of features to

maximize space on the users screen. The canvas that holds the WebGL should be the main element on the page, and the options and menu should be placed in the most optimal location to not effect or restrict the view too much. Alongside considerations for this, there should also be a use case for the GUI, so that it is possible to group functions, and place options together making the application easier to use. Although the primary objective is not to concentrate too much on the front-end in this project, it should be made sure that it is attractive and well laid-out. Because FDD has been chosen, it is better to have a good looking and less functional program rather than more functions but unusable program due to the front-end.

2.3.1 GUI Use-case

spot As mentioned earlier, use-case would be a good way to group initial front-end functionality. For my application, creating a use-case means it can then be decided which group of options go where on the site. The following use case in figure 2.3.

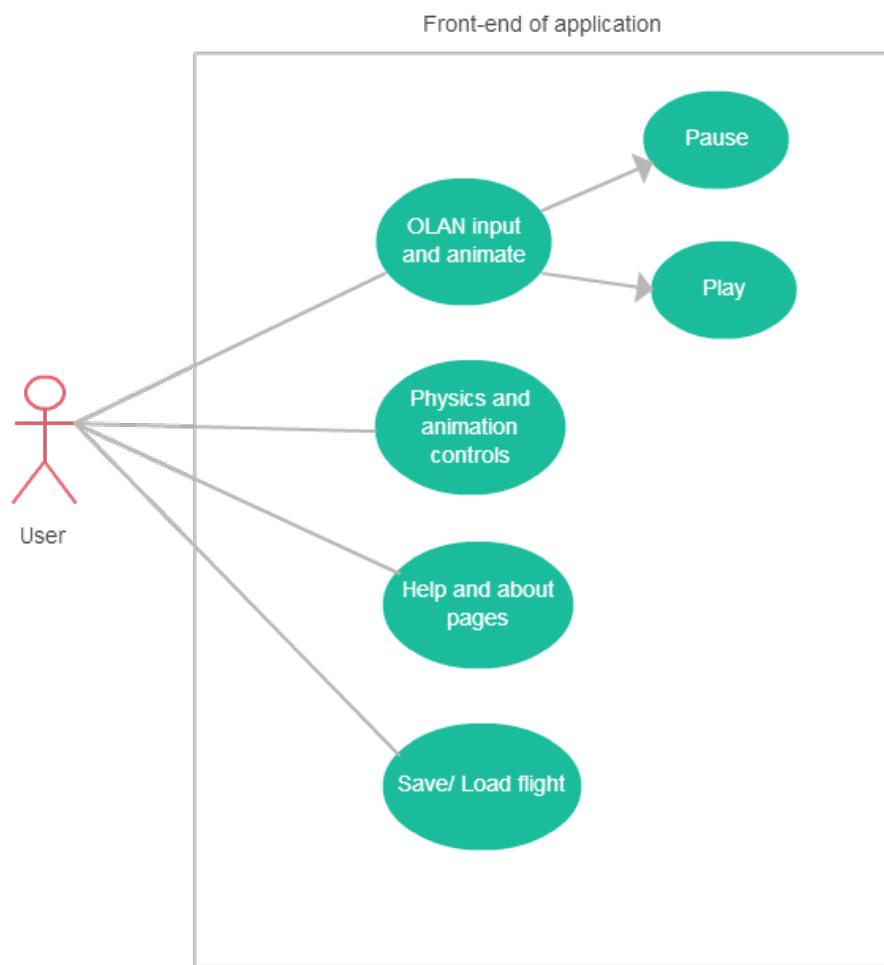


Figure 2.3: Use-case diagram detailing user options, the canvas and OLAN input.

As you can see, there is the possibility of four sets of options each of which could have its own category in my GUI. By grouping them this way, it means the user could find the options much easier, and give a less unorganised feel to the application. Starting with the OLAN options, there

are going to be two primary functions that should be required here: entering the space separated OLAN and parameters, and a drop down to allow for input from a list. The latter of which should be in its own section in the menu, whilst the first will be a simple input box at the top of the web page about the canvas. Following this, there will be the animation controls next to the input box, to begin and pause playing the flight. I have chosen the input and play button to be at the top of the page not in a menu because these will be the primary elements of the application and should be quick to use.

As for other menu pieces, there is the possibility that physics and animation related options will be available, and should be listed together in a menu. These options contain some of the extra features mentioned in my analysis, such as wind and other items that will be implemented after key functionality. Therefore, although there will be a menu for these, there may not be very many options depending on what work is complete at the end of the project. The next menu that would be required, and the last one, is the saving and loading of projects. There should be four or five options here, for saving in both formats, and loading in both formats. One of which will contain a file browser for selecting JSON files that were exported from the OLAN flight input, and then a button to begin uploading and inserting into the application. An idea alongside this would be to have some form of status bar when the application loads up a flight, depending on the size of the file. If the save files only store OLAN input and not rendered vectors, the need for the loading information would not be required.

An additional set of options, but rather extra pages will be the about and help sections. Instead of placing these alongside the other menus, these can simply sit onto the top menu bar next to the OLAN input box. Rather than options, these should be simple HTML pages that give the user some insight into the application.

It was mentioned earlier in the report about using the lazy initialisation pattern, and simply disabling options could help towards this. For example, a user may try to enter OLAN and press the play button before the manoeuvres have loaded up causing an error. An idea could be to make the input box disabled by default, and then enable it from the back-end once the application is fully loaded. To help understand this, a flow chart in figure 2.4 shows what the user should be able to at what stage in loading of the application.

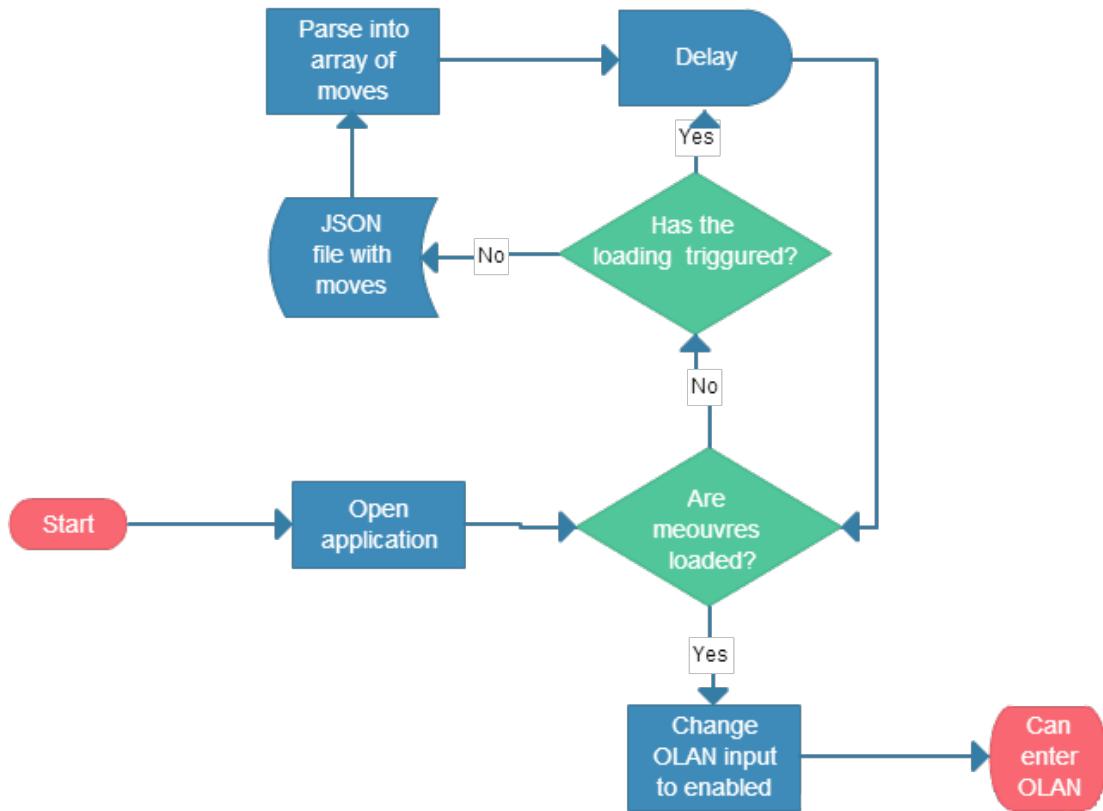


Figure 2.4: Flow chart showing how the application should run in accordance to loading up and then allowing the user to begin using various functions.

As you can see from the flowchart above, the input box that allows OLAN entry will be disabled until loading of the aircraft model, manoeuvre JSON file and setting up of the various controllers. Options should already be enabled when loading up, as back-end checking can be performed separately to lighten the front-end. In any case, because listeners are set up last, any options in the GUI will not function until the rest of the application loads up anyway.

2.3.2 CSS and wire-frame design

Following finding of all the required use cases on the application, it is now possible to create draft wire-frame designs for the front-end of the application. Because it would be ideal that the web page works on both mobile and desktop, a CSS library could help to create this easily and quickly. After considering the two main CSS libraries used on the web (Zurb Foundation [14] and Twitter Bootstrap [?]), it was decided that the first would be more appropriate for the application due to some built in functionality that could be useful to the application. The functionality that was found to be most useful was an off-canvas menu. This menu allowed content to be hidden to the side of the page, and then when opened pushed content on the page to the right. This is especially useful for this application where getting maximum space is best for the canvas. In addition to this, the menu allowed for stacking, so it is possible to add sub menus that also push in from the left.

Alongside the menu feature, the general appearance and flexibility of the library will give the application a more attractive and professional appearance, for example replacing check-boxes for

options with sliding switches. To give a more detailed insight into the idea of what the application could look like using this library see the following wire-frame in figure 2.5.

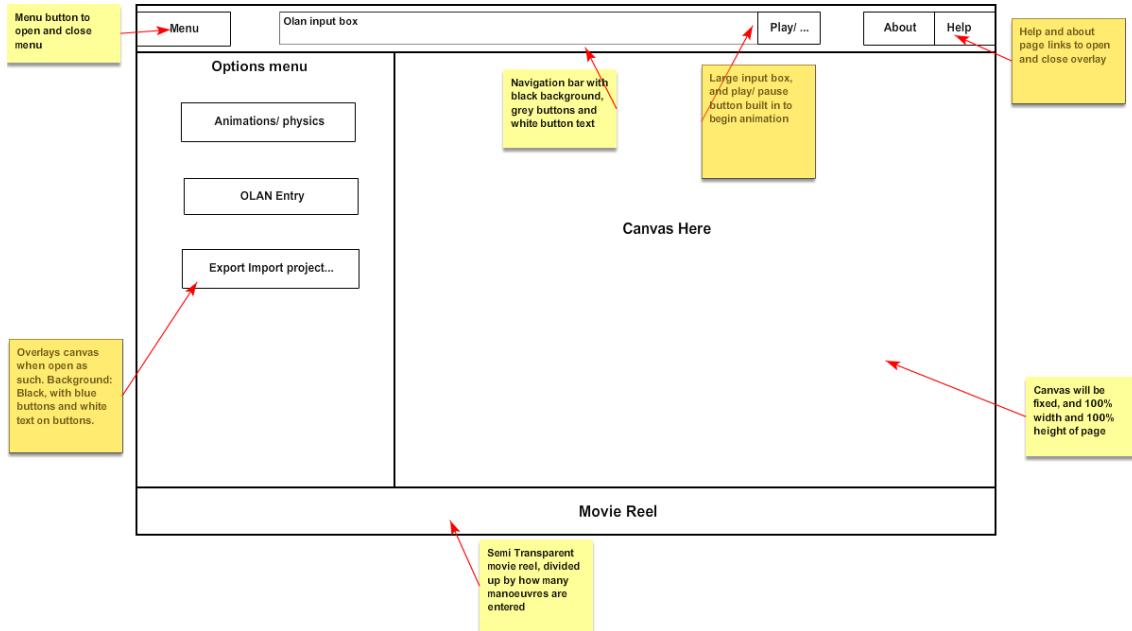


Figure 2.5: Wire-frame design detailing desktop view of the application, including the open menu for the 'off canvas feature'.

As well as the desktop design of the site, a design of the proposed mobile view can be found in the following figure 2.6.

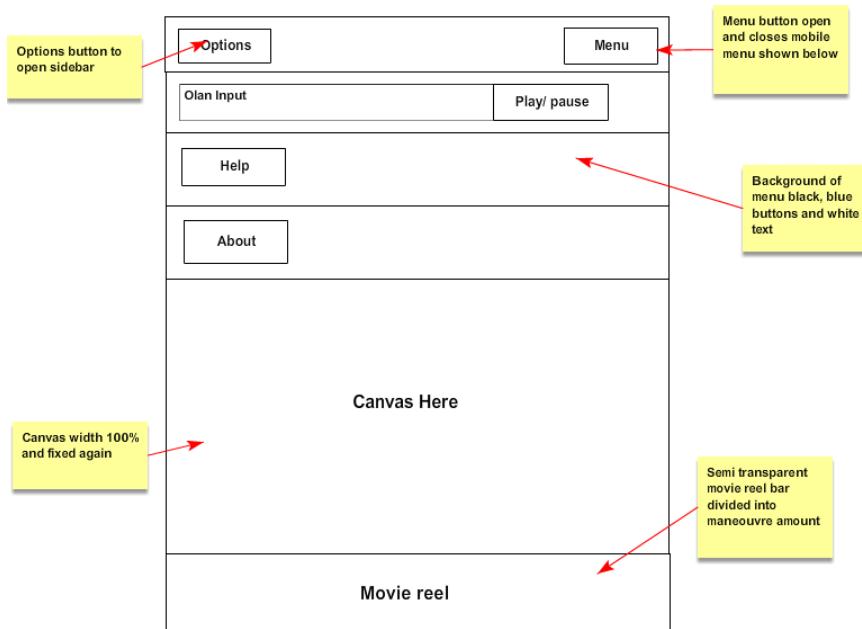


Figure 2.6: Wire-frame design detailing mobile view of application with navigation bar in minified version.

It can be seen that the mobile view follows the same style as the desktop version in the fact that menus and other elements are kept hidden away while showing the canvas for maximum space usage. Again, the off-canvas menu will be used to hide all the option menus, whilst the navigation containing the OLAN input and play button will be hidden at the top of the screen like a standard mobile navigation bar. Items like the about and help pages will take up the entire page on mobile to make text easier to scroll through, whilst on the mobile the help pages will only take around 50% and overlay the canvas.

In addition to both wire-frames and their respective descriptions, it should be pointed out the reasoning for colour scheme choice and other accessibility issues. It was decided that the black background with white text was best for readability and a cleaner look. If the colours were flipped, the application edges such as the navigation would not stand out enough, and finding the edge of the canvas harder. Because the foundation library is being used, it is possible to change the colour scheme from its default blue to white easily through its custom download page. It will also be endeavored to create the site as responsive as possible, with menus and navigation elements correctly scaled on different sizes systems. This should mean the application should appear as well as a tablet than a mobile.

2.4 Planned use of services

Throughout the development of the application, it is planned that a range of tools will be explored and used to help make implementation, testing and documentation more efficient. The first of these is Github [15], which will give two features: allow version control of the development work, so to keep it safe and secure, and to provide a free web hosting service. This can be achieved by creating a 'gh-pages' branch in a repository as seen on the Github site [16], and then any files in that branch can be navigated to on-line. This will be especially useful for my application, as after each commit, the results of changes will be reflected on this branch's web-link. Github will allow work to be carried out on any machine, and to revert any changes that may produce errors when implementing features.

Working alongside Github, a service called Travis [17] will be used to test the application. The Travis continuous development server will allow the application test files to run on each commit to the Github repository. It works by firstly linking to Github, then by reading a file in the repository containing instructions of running any test files that exist. Travis brings with it lots of useful features, as it is possible to run a wide set of different languages and libraries from the Travis file in the repository. The testing library toolkit that will be used in the development of the application will be JUnit, alongside a framework called Jasmine [18]. JUnit acts a lot like any other unit based test, allowing a set of tests to be run after another testing different scenarios across an applications functions. Jasmine will allow for cleaner behavior driven testing of JavaScript code and will run headless, meaning the application can be tested without loading up the web-page.

A library that will be used during the implementation stage will be the JSDoc library [19], which will help when creating organised documentation for all the JavaScript methods and variables. It is important that each JavaScript module implemented has easily understandable comments to help future development for times when it may have become unclear what one part of the code is doing. In order to keep up with documentation, with each function that is implemented, relevant JSDoc animated comments will be added at the same stage. This means of documenting will make it effortless to create the documentation when the application is complete, by simply running the

library from the command line to generate the documenting HTML pages.

Another technology that will be used to implement the application looks towards the front-end. Rather than standalone CSS for styling the web-page, Sass [20] will be used. In order to use Sass to style a page, it must be complied into its primitive CSS form before adding it to a site. Therefore, while developing the site locally, a software known as Koala [21] will be used to automatically compile the Sass file holding style information after each change to any Sass files. The main reason for using Sass follows the rule of keeping the application as maintainable as possible, and easy to understand for future referencing. As Sass uses a good layered structure, it enforces this easy readability.

The final supporting service that would be useful to have to increase productivity during development is a local server application. As I have had past experience using a program called EasyPHP [22], this will be used when performing coding. The main advantage of using this is that it will be possible to edit and see the site live, rather than having to commit each time. The local server is needed due to issues loading JSON files dynamically through JavaScript which is not possible to do through simply opening the HTML file in a browser.

Chapter 3

Implementation & Testing

Following from the analysis and design of the application, it is now possible to start implementing the features produced from investigating the requirements earlier in the project. As it was already stated that the project would be running under an FDD means, rather than having two sections (one for implementation and one for testing), this section will show each feature one-by-one. By following this scheme, the feature priority list will be utilised to ensure the more important features are done first, and that they fulfil their function before moving on. However, this should not hide the fact that issues are possible when implementing and any such issues could hamper further development. If this does occur, the issues and reasons will be explained, followed by what judgement was made to continue on with development of the application as a whole.

3.1 FDD project approach

As mentioned, the feature driven development methodology will now shape the approach that is taken to create each feature. Although design has already been completed for all features, there are still elements to each iteration that should be outlined before proceeding. The best way to plan this is to give a brief list of steps that will be performed in each iteration. This is as follows:

1. Give a re-cap on the requirements of the feature
2. Provide a walkthrough of implementation (with code samples)
3. Show tests used (JUnit, usability tests) and results
4. Describe and differences with the feature and original design
5. If there are any issues, describe and give explanations
6. Give a progress log on the item, and provide dates and length of completion for use in accordance with the Gantt chart originally constructed

It should also be noted that once the duration of the time in the project for the implementation and testing stage is complete, a burn down and progress report will be possible to generate and will be done so. This will give a good idea of exactly the status of the project (with a calculated percentage), to see how much work was done compared to estimates, and to find out how successful the project was overall.

3.1.1 Iterative Implementation

Before the iterations begin, it should be mentioned that although the order of features to be implemented were decided in the analysis, in order to implement any back end things and allow for proper testing (usability testing), some basic GUI must be created first. Because of this, this will be the first iteration, followed by creation of the JSON manoeuvres. Following these iterations, it will then be much easier to test future features created. Without a basic GUI and canvas, it will be impossible to see the effects created whilst adding features such as drawing flight paths or moving cameras. After these two moved features are complete, the prioritised list will run as planned initially.

3.1.1.1 Feature 1- Create a basic GUI

The first feature that was to be implemented was less of a feature but more a requirement so the other features could be then created. The GUI or front-end of the application is important to be created, otherwise testing future WebGL features will not be possible. The requirements of the front end were:

- A responsive front-end that is mobile compatible
- Have hidden menus that can push in from the left
- Have basic inputs such as the OLAN input, and menu options
- Provide a help and about page that overlays the application web page

The first task that was performed to create this task was to create a minimalist layout matching the wireframes designed in chapter 2 of this report. The first action performed was to get the latest Foundation [14] css library file, and then to create a layout using the features from the library. The navigation was made using the 'navbar' markup, which helps stick the navbar to the top of the page, alongside automatically changing into a mobile pull down menu once the screen size reaches a certain amount. Then using other mark-ups provided by Foundation I was able to float the various navigation bar buttons to their respective sides.

The most important part of the GUI which has now been implemented is the off-canvas feature menus. The code for this can be seen in section 3.1 of appendix C. The way that Foundation helps create this off canvas option is by wrapping everything in one div, then separating the menu and content in half. When a user then pushes the designated button with the wrapper id, the entirety of the wrapper is pushed to the right to display the menu and less of the content. The about and help pages were also added successfully, with the addition of another library called Modernizer. This library allowed for a clean effect to overlay the information on top of the webpage.

Once the basic HTML elements with relevant ID tags were added to the index page of the application, it was then possible to add some custom style to the page. As mentioned in the design stage, it was possible to download Foundation modified with the colour scheme designed for the application. Changing backgrounds to black and text to white was very easy, and then added to the site. The colour and design of the page can be seen below in figure 3.1.

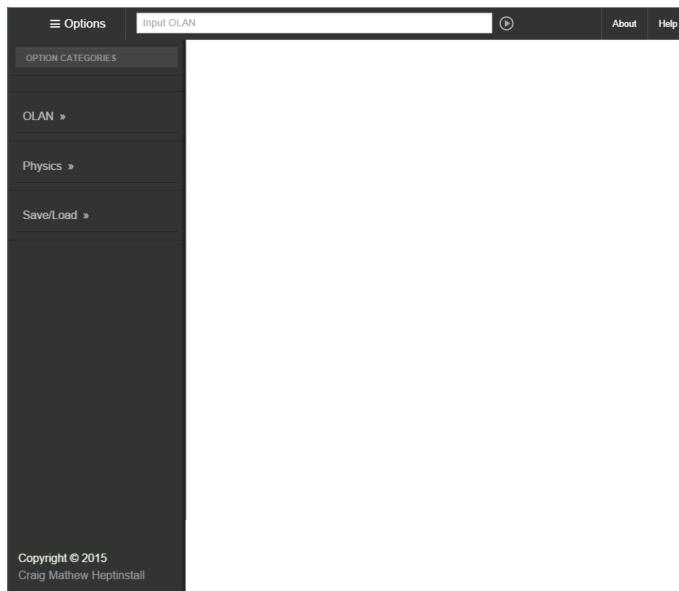


Figure 3.1: Screenshot of GUI created with Foundation, JQuery and HTML markup.

As planned, Sass was used to style specific elements of the page, such as the width of the OLAN input box, and to change text placement on the help and about pages. Setting up the Sass with Koala was easy, and upon each save of the file, CSS was compiled.

One final mention goes to the creation of the WebGL canvas. To create the basic canvas, only a few lines of JavaScript was needed. To create it, a renderer was created using the 'THREE.WebGLRenderer' object. Because this is early in the implementation stage, it was created without RequireJS as the architecture is only one JS file. Once the object is created, JQuery appends the '.domElement' of this object to the container div created in the basic HTML. For future features, the WebGLRenderer object will be where any objects such as flight paths are added to.

As for testing this feature, because very little JavaScript was required to be coded here, usability testing was the only way to check for the completion of what was required. The test results can be found in Appendix D under section ???. Overall, this feature has been made to 100% of the requirements laid out, and matches the design very well. Therefore, the progress tracker for this feature can be shown as:

Start	End	Duration	Progress	Comments
09-03-2015	10-03-2015	2 Days	100%	Complete, though once new features are under way, RequireJS will be utilised and the WebGL Canvas code will be moved.
11-03-2015	11-03-2015	1 Day	100%	Testing complete, usability table created and tested on mobile and desktop devices.

Feature overall progress: 100%

3.1.1.2 Feature 2- Convert OLAN and Aresti to JSON form

The second feature, which was of high importance to the rest of the application was to create OLAN interpreted JSON which would give instructions on how to construct each manoeuvre. Because this was quite thoroughly planned and then designed how each instruction was going to be broken down, the time designated for this task felt generous. Simply using the template created in the design, it was possible to fill in each instruction part by part through each OLAN manoeuvre. An example of some instructions can be seen in Appendix C in section 3.2.

The only time consuming part of this feature was the sheer amount of different manoeuvres that had to be converted. For this reason, not every OLAN letter was created. The reason for this is because some moves are possible to be created from others, so further in development, more will be possible to be added.

To support this feature, more JavaScript code had to be added to allow the JSON to be read into an object to begin drawing moves onto the canvas. Because the code would be in a different module to the WebGL canvas that was created in the previous feature, RequireJS now had to be added to the source code. To do this, the RequireJS library was added and initiated in the HTML page of the application as shown here in Appendix C section 3.3. Then a new module was created, named 'Dataparser' as designated in the design. The method for converting the JSON can also be found in the data parser module shown again in the appendix under section 3.4.

Once the coding and converting to JSON was complete, testing the code that performed the conversion to an object was required. By using JSUnit, it was possible to call the method that was created to instantiate the manoeuvres array, and pass in some raw JSON string, then compare to what the result should be. The JSUnit code can be found in appendix D, section 4.2.

Like the last feature, no changes from the original design were performed, resulting in a good standing of time compared to the Gantt chart for the project. The feature progress can be expressed as percentages and final comments added again.

Start	End	Duration	Progress	Comments
12-03-2015	16-03-2015	5 Days	80%	JSON created to represent most manoeuvres, for that reason it is not an exact 100% progress. JavaScript code and RequireJS functionality added to support the rest of the application.
17-03-2015	18-03-2015	2 Days	100%	JSUnit tests created and passed successfully, showing JSON matches up with manoeuvre array object.

Feature overall progress: 90%

3.1.1.3 Feature 3- Creating a scene with terrain and lighting

The third feature, which is another to be brought forward ahead of flight path creation is the terrain and lighting effects. The reason for creating this before the OLAN construction is the same as creating the canvas, because without an area to see where the paths are drawn onto, the effects of that feature will be hard to test if it is working. Especially terrain, where the need for this is to see where the X axis of 0 will be. Without this, when implementing checks such as if the path

would hit the ground would be harder to test. Again, the three.js library was of great use here, especially with built in methods to allow for lighting to be added.

In order to have started this feature, another module was added to the architecture named 'TerrainHandler', which was where both lighting and ground was to then have their methods of adding to the canvas. This feature was found to be the shortest of the features, as it only required two simple methods. Rather than passing the renderer canvas object to this module to add the ground and lighting, it was decided to simply make both methods like factories to return the ground and lighting objects to where they were called from. In the case of where this module was called from, the main class where the canvas was created seemed the best place to call each method, as everything to do with setting up the scene could remain together and be easily understood or developed further.

As for testing, by keeping the terrain and lighting in their own module also helped towards testing the new methods. Two very simple tests were made using JSUnit to check the returns of each method, and both were added to the Jasmine instructions under each Github commit through the Travis builds. As with other tests, the test cases can be found in the appendices under appendix D section 4.3. This feature was fairly straight forward to implement and test with help from the three.js documentation site, therefore the feature was done in less time than originally planned.

Start	End	Duration	Progress	Comments
19-03-2015	20-03-2015	1 Day	100%	Completed module for terrain and lighting, linked up to main module.
20-03-2015	20-03-2015	1 Day	100%	Two JSUnit tests created, and passing successfully.

Feature overall progress: 100%

3.1.1.4 Feature 4- Cameras

The final preceding feature before the creation of OLAN paths that was decided should be ready were the various cameras that will look around the canvas. The initial requirements for cameras was to have two different views: one for navigating around the canvas, and the other for being onboard the flight path. At this stage of the project, creating the second was only possible to a certain extent, because without the construction of the paths yet, getting the location of where the onboard camera should be is not possible. Therefore, for this requirement both cameras were created, with the exception of the onboard camera where functionality is currently restricted.

To start, another module was created for the 'CameraController'. Because it was best to keep the cameras completely separated from other parts of the application to reduce decoupling, both camera objects were stored in this module and provided get and set methods for calls from other modules. An initiator method was created for use from the main module, to create both camera objects at start up. Both these objects were again available from the three.js library as 'THREE.PerspectiveCamera'. Then by passing the module around to any other modules, it will then be possible to update and retrieve each camera object again. Alongside the camera creation, a controller for changing the angle of view, zoom and coordinates was created as another module.

This module, which has been called 'CanvasController' as designed in the module diagram, adds event listeners for dragging the mouse around the canvas, keyboard presses to move up and down, and scroll for zooming. Then by getting the new updated values of view-point angles, these are

passed back to the camera controller which updates the values of each camera. By repeating this in a render loop in the main module, the updating of cameras is done several times a second, to make movements appear fluid. To help this, the code was refactored several times to ensure that the amount of work done by the browser is as little as possible on each loop of the animation.

As for testing, alike the first feature where usability testing was the primary and sole means of checking the implementation fulfills the feature, JSUnit testing was not used. Usability testing was found the best option here, and results can be found in appendix D section ???. During testing, it was found that there was an issue that when rotating the camera around the canvas, it seemed to always rotate around the point (0,0,0). This meant that navigating the canvas was slightly harder than it should have been. This was fixed though, by simply moving the camera rather than the scene in the canvas. As for the camera that was designed to be as on board view, a model airplane was added to represent the camera so it could be seen on the canvas. Although it does not move yet, the plane loads up at the start of the application and can be placed anywhere on the scene. You can see the plan on the canvas at point (0,0,0) below in figure 3.2. The plane model is constructed from a JSON file from a page on the web with thousands of free models.



Figure 3.2: Model of plane loaded up using JSON and three.js object loader. This plane will represent the onboard camera, and fill be the object that will be assigned locations along an OLAN path to show flying the flight plans entered by users.

The progress of the task is shown below. This feature was done exactly to its requirements in the design, and the modulation of code is still being followed.

Start	End	Duration	Progress	Comments
21-03-2015	23-03-2015	3 Days	100%	Both cameras now usable, with constructor methods, and update methods. On board camera ready but not linkable to rest of application until animation of flight paths is ready.
24-03-2015	24-03-2015	1 Day	100%	Usability tests complete, one change made to code following a failed test.

Feature overall progress: 100%

3.1.1.5 Feature 5- Construction of 3D paths from OLAN

The most important feature, drawing shapes onto the canvas using the instructions read in from JSON files was next to be implemented. The main requirement here was to draw smooth shapes that match the Aresti shapes as accurately as possible, also bearing in mind that they should be linkable.

Using the array of manoeuvre objects imported from JSON in the first feature implemented, creating shapes based on these was a matter of building up an array of vectors with each new vector being copied from the previous and then having the next instruction take effect on it. To do this, under a new module 'ManoeuvreController', a method was created that firstly found which manoeuvre was to be built, then loop through the instructions of that manoeuvre creating it vector-by-vector. The means in which each manoeuvre is constructed from the OLAN JSON can be seen in appendix C section 3.4. On each iteration, once the amount in each direction (x,y,z) has been found,

these values are applied to a new vector with values copied from the previous one in the array.

Once vectors were created, they were passed to a three.js object which was a spline curve. A spline curve is one that interpolates along all its points to create a smooth shape. Once points are added to this object, it can then be added to the canvas. Upon initial usability testing of this, it was found that many of the joins between points were well-off center, due to the interpolation being too large. This was down to the fact that with each turn instruction being a 45 degree angle, meant that the change was too sharp from a straight to curved line, so the straight line was affected too much. To fix this, by dividing up the 45 degree into smaller pieces meant that interpolation was needed much less, therefore the curves appeared smoother and had much less effect on any preceding straight lines.

Another big issue that appeared during testing was that any manoeuvres that had a change in angle through a turn always seemed to revert back to a straight line, meaning the rest of the manoeuvre was out of sync and did not look correct. This was an especially time-consuming bug, and an example of it can be seen in figure 3.3. Because finding out the problem and attempting to fix it was eating away at other feature time, the issue was left for the time being, in order to complete other key features outlined in the requirements. In order to ensure this issue would not remain at the end of the project, a strict time limit was set on the next features to allow for a good amount of time to fix the issue here.

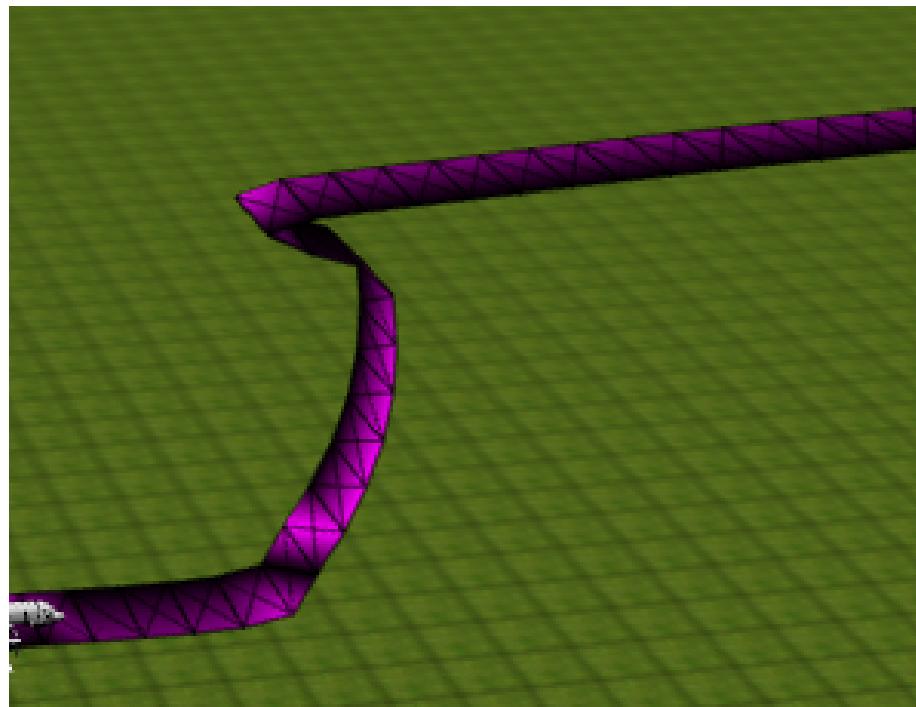


Figure 3.3: Figure illustrating that where the vector is turning right to complete the diagonal line up, it is becoming flat with the X axis rather than curving to the correct angle.

At the end of the iterations in this report, this issue will be re-evaluated once the fix is in place. Other than this issue, the rest of the feature was implemented well, and entering OLAN now drew fairly accurate shapes. Again with testing, usability was the primary case here, because checking for correct spline curves created would be too time consuming and difficult with the use of JSUnit. The overall progress of the task at this point of the project is as follows:

Start	End	Duration	Progress	Comments
25-03-2015	05-04-2015	2 Weeks	75%	Although OLAN now is represented by shapes and they link up correctly, the issue mentioned os straight lines being drawn on change of angle remains. This will be fixed fully after implementtaion of other tasks.
06-03-2015	08-03-2015	2 Days	75%	Usability tests complete, changed step of angles from 45 to 15 to get smoother interpolation. More tests will be required after fixing the issue.

Feature overall progress: 75%

3.1.1.6 Feature 6- Animate OLAN flight path

Although there was an issue imnplementing the drawing of flight paths in the previous feature, the functionality that was complete allowed for the animation feature to be done. Three.js provided a good means of getting points along the spline curves to a percentage of the length of the shape. For example, by using the method 'getPointAt' with 50 as the paramter value gave the return vector value of the point on the line halfway. By doing this iterativly using the render loop used in the cameras meant a value could be incremented over time, thus getting points along each curve. Each time a point was received, the on-board camera could then be set to the position of it, and by doing this as fast as the render loop meant that a smooth animation was created. Again, a new module was created 'AnimationController' with various methods for playing and pausing the animation, and setting the distance of time between each point received in order to speed up or slow down the animation.

With each curve being stored in an array object, meant if linked up manoeuvres were being animated along, once one manoeuvre had reach 100% for getting points along its line, the next manoeuvre would be selected from the array and the get-point percentage reset back to 0. By doing this, a flawless smooth cross between manoeuvres has been achieved.

As for testing this feature, more JSUnit was used, but mainly for the getters and setters of the controller. For instance, checking that the pause, play and speed setters worked correctly required simple test cases that checked the private variables holding the values were correct. Alongside these tests, usability tests were also completed becuase of the new buttons in the GUI representing the play/pause and speed options. Both tests can be found in the appendices under section 4.5. Following the completion of this feature, the progress can be reported on.

Start	End	Duration	Progress	Comments
09-04-2015	10-04-2015	2 Days	100%	Animation is now possible through all manoeuvres in a flight, alongside options to pause, play and change speed.
11-04-2015	11-04-2015	1 Day	100%	Simple JSUnit and usability tests performed to check values from GUI are being sent to the back-end correctly.

Feature overall progress: 100%

3.1.1.7 Feature 7- Exporting and importing routines

The last of the primary features planned for the application was the saving and loading of flight plans. Because up to this stage the project time was behind, and time was required to fix other issues, it was decided that the initial format of saving the OLAN input directly would be quicker to implement than the rendered vector values.

To start this, a final module was created for the handling of JSON files and local storage, and this was then linked up to the HTML handler module to get the user's current OLAN input value. Four methods were required for this:

1. Exporting to JSON- This method makes a simple JSON file with one object contained ("OLAN" :olan string)
2. Importing from JSON- Allowing selection of a user file from the browser, and selecting the OLAN from it, then entering this into the input box to be rendered.
3. Exporting to local storage- The same format as the JSON, but saving in the browser. See appendix C section 3.5.
4. Importing from local storage- same, should be automatic when application loads up

In addition to the requirements that were implemented, it was decided that user should be able to choose whether automatic backup of the entered OLAN would load up at the start of the application. Because of the observer pattern that is being used throughout implementation, it took a very short space of time to add this feature to the HTML handler listener.

Testing involved some JSUnit again, where test data was entered and checked against the resulting JSON and local storage objects. The tests can be seen in appendix D section 4.6. Up to this point and for future tests, it was ensured that the Travis builds were passing when running all the JSUnit tests through Jasmine. This feature was completed quicker than the allocated time due to the research into Local storage previously, and therefore helped gain back some valuable time for other features.

As for any future development, it was mentioned that it could have been better to export the vectors directly, but the space in files and in time was considerable here therefore helping towards efficiency in this part of the project. The ending progress of this feature follows in the table below.

Start	End	Duration	Progress	Comments
12-04-2015	14-04-2015	3 Days	100%	Both importing and exporting of JSON and local storage complete, with an additional feature of giving the user the option to auto save when they enter any OLAN.
14-04-2015	15-04-2015	2 Days	100%	Implemented JSUnit tests linked to the Travis build, testing the format and content of exported flight plans.

Feature overall progress: 100%

3.1.1.8 Feature 8- Adding further GUI options

By this point, quite a few of the features installed into the application had the possibility of allowing the user to interact with a range of different parameters and options. For example, with the spline curves, there were options such as the smoothness of interpolation (the number of extrusion segments), options to switch between cameras, and the scale of flight paths amongst others. As with other options that had been added along the way with some features such as speed and auto-save, linking each to the GUI was fairly straightforward, with all that being required was to create an ID for the HTML element to be an option, and then listen for a change or click using the HTML handler, finally calling its respected method.

The options that were added can be shown in a comprehensive list with reasonings for each.

- Extrusion segments- The smoothness of paths, where the more the user chooses the more segments each curve is made of. This can help users with less powerful systems, as there is less to render with less segments.
- Onboard view- an option that changes the current camera view to one where the user will view the canvas environment from the point of view of the front of the aircraft. This gives the feel of actually flying the OLAN path, which could be helpful to users such as pilots who may want to learn a routine from the inside of a cockpit.
- Scale- The scale of all the OLAN paths drawn, defaulted at 1, but can be increased in multiples. Useful if the user wants to fit more onto the canvas, or wants to get a larger better view of a move.
- Radius segments- The initial line that is drawn to show the manoeuvre may be changed to have more sides. For example, the default being 2 means a flat ribbon is drawn. 1 would be a single line, whilst 0 would make the path invisible. This would then allow the user to have an aircraft fly the path without any shapes behind it.

An additional GUI feature that was added here was the check for the loading of the OLAN JSON file before activating the input box in the navigation bar. As mentioned in design, this box should not be editable until the application is fully loaded, so a method was created in the HTML handler to activate the box when needed. This method was called by the camera controller module once the model aircraft was loaded, as this was the last and largest file to load.

As with any GUI interface features, usability testing was the best option here, and therefore each new option added was tested after being created. The tests checked if the proper function was called and had an effect on the application as was planned. The tests follow the previous ones in appendix D under section ???. It should be noted that the tests were done both on mobile and on desktop, to keep up with the added requirement that the application should be usable on both platforms. The tests ensured that all the options worked well, and worked when they were supposed to. The following table shows this feature's progress.

Feature overall progress: 100%

3.1.1.9 Feature 9- Flight 'movie reel'

The movie reel feature was one that would be added if there was enough remaining time, though in this case it was decided that it would be implemented to a set time limit. The aim of the reel would

Start	End	Duration	Progress	Comments
15-04-2015	15-04-2015	1 Day	100%	Options added to reflect changes in various aspects of the application, alongside the lazy initialisation check added to the OLAN input box.
16-04-2015	16-04-2015	1 Day	100%	Usability tests performed on desktop and mobile browsers, all passing after multiple times refactoring.

be to show the user what manoeuvre is currently being flown in the animation, by having images or the manoeuvre from the OLAN input box converted to simple 2D Aresti, and then shown along a reel with current progress across them in relation to the entire flight. To start this, it was firstly considered that using the logic from the manoeuvre controller that built flight paths could be used to the same effect for drawing the movie reel figures.

Upon starting this though, it was found that many aspects of the feature would take too much time to implement, and that they could cause performance issues for the user. The first of these was that if the application was going to draw mini figures at the bottom of the page in a fixed location, it would mean having to create a set of mini-canvas to interact with, rather than using the main canvas and drawing at the bottom of it. Each figure would thus need its own canvas, and after so many are added, this can dramatically affect the browser's performance. Another issue found when looking how to create the reel was the complex style of Aresti figures required. For example, creating arrows to represent rolls, and for annotations to be generated in the correct places alongside curves would require detailed mechanics in the back-end in order to produce them. This and the first issue was decided would take up too much development time, thus pushing back the remaining features left to fix and implement. Therefore a compromise was made, where the reel has a range of images of the Aresti figures loaded and put in place alongside each other. These images have already been created using the OpenAero application, so saving these to a folder in the application was easy, and each was saved with the name of the OLAN letter.

The movie reel was incorporated into the HTML handler, as this was a dom-element related item and required JQuery to manipulate the space at the bottom of the canvas. Then, added to the listeners in the HTML handler file was an event where when the user enters an OLAN letter, and this is found in the manoeuvres array (to ensure it is a real OLAN notation), an image representing the Aresti figure with the letter for the filename was loaded into a div floated at the bottom of the page where it was appended. Although this means of creating the figure was not what was initially planned, due to time constraints this means of creating the feature ensured it was done as quickly as possible, but also to a good standard of presentation that fit well with the rest of the application. It can be seen in figure 3.4 the reel with some figures loaded up, with the current animation progress highlighted over the manoeuvres.

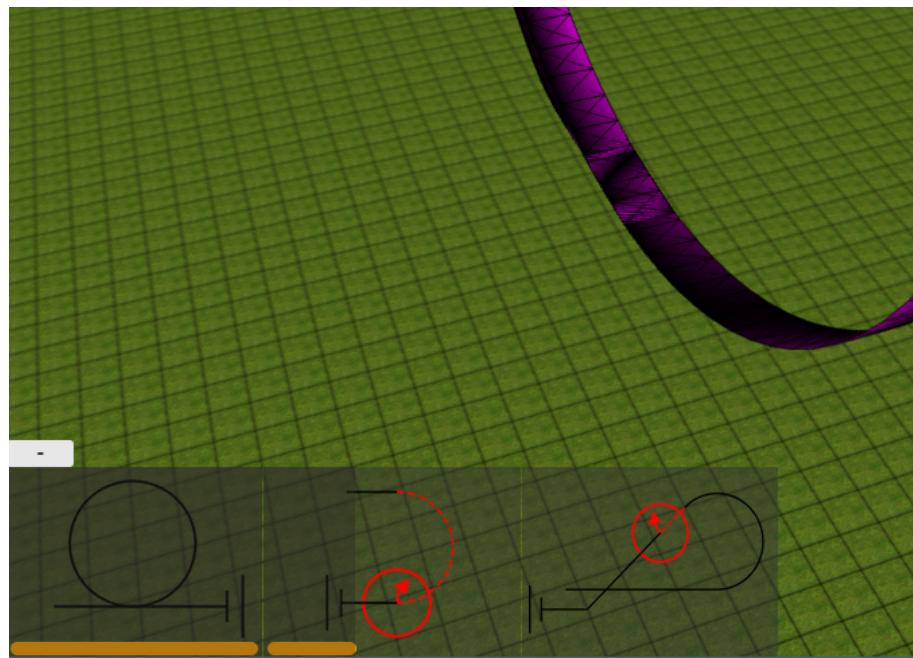


Figure 3.4: Movie reel overlaying the bottom of the canvas, with some figures loaded after the user typed in a string of OLAN.

Figure 3.4 also shows the animation progress, which was achieved by simply getting the current value of time which was used earlier in the 'getPointAt' method, then getting the percentage of this for the amount of moves entered. After this, a simple div was placed again using Jquery over the width of that percentage to achieve a darker coloured semi-transparent status bar. During tests of this feature (usability), one such issue was found that was caused when more than five manoeuvres were entered, the remaining Aresti figures were not present on the page, as it had been split into five sections. This means at this point of the project only the first five manoeuvres have their animation status shown. This is another feature that is hoped to be fixed before the end of the project implementation stage.

Start	End	Duration	Progress	Comments
17-04-2015	19-04-2015	3 Days	50%	Movie-reel implemented using images rather than constructing, though they show correctly the Aresti figures when the user enters OLAN. Issue found that requires fixing, concerning having more than five manoeuvres available on the reel.
20-04-2015	20-04-2015	1 Day	100%	Usability tests complete, found the issue using them, and will test again once the issue is addressed.

Feature overall progress: 75%

3.1.1.10 Feature 10- Parameterisation of OLAN input

The final feature that was added to the implementation Gantt chart was the one that would allow users to enter both parameters to OLAN notation (such as roll amount, length of entry and decent or others), and to tell the manoeuvre controller to move the start location of the next manoeuvre so many co-ordinates along the X or Y axis. Current progress on this feature after moving on in this stage of the project was that the second part functioned, allowing users to enter elements to move the start position of the following manoeuvres. However, as with some other features being implemented, it was felt that the time required to add the various prefixes and postfixes for each and every OLAN shape was too consuming on the overall project time. As with features where it was taking too long for specific parts, this will also be left in accordance with how the Gantt chart stands for time left for implementing and testing. The function for calculating the parameters was added besides the existing method that reads input from the OLAN box, then creating vectors to create a path between a previous and next manoeuvre.

For the part of testing the half of the feature that worked, usability testing was the best approach here. In appendix D section ?? the test table can be found, and shows the feature passing for this aspect. By attempting to keep the workings of the application as similar as possible as to how OpenAero handles parameters keeps consistency for users across the web, and to keep consistency in the OLAN language. While testing, it was found that the input of the user should be heavily restricted and strictly typed, ensuring the user types the exact format that the parameters should follow. After testing and evaluating the way that the OpenAero application handles this, it was decided the format should also be '(x,y)' where it is space separated from other elements, and where x and y can be either negative or positive.

Following the completion of this particular parameterisation, the progress of the feature can now be reviewed.

Start	End	Duration	Progress	Comments
21-04-2015	23-04-2015	3 Days	50%	Parameters for moving start position of next manoeuvre ready, but individual OLAN manoeuvre parameters not implemented yet.
24-04-2015	24-04-2015	1 Day	50%	Can only test first half of the feature up to this point, will implement tests if second half is complete.

Feature overall progress: 50%

3.2 Final status and progress

At this stage of the project, it is now possible to get a grasp of the overall status of the application concerning implementation and testing. Using the progress reports from each feature, a general idea of the entire progress will combine all the completed features and then allow the creation of a burn-down chart to compare the current status with projected status. Although Burn-down charts are not FDD, it was chosen that the Scrum approach here would reveal more information around what work still needs to be completed, and when by. In order to create the burndown chart, the overall percentages of progress for each task should be collected. The list of features is shown in table ??.

Feature	Start	End	Duration	Progress
1	09-03-2015	11-03-2015	3 Days	100%
2	12-03-2015	18-03-2015	7 Days	90%
3	19-03-2015	20-03-2015	2 Days	100%
4	21-03-2015	24-03-2015	4 Days	100%
5	25-03-2015	08-04-2015	2 Weeks 2 Days	75%
6	11-04-2015	09-04-2015	3 Days	100%
7	12-04-2015	15-04-2015	5 Days	100%
8	15-04-2015	16-04-2015	2 Days	100%
9	17-04-2015	20-04-2015	4 Days	75%
10	21-04-2015	24-04-2015	4 Days	50%

Table 3.1: Collected statuses of all features required, with their duration and percentage complete. Note: the dates shown the task was completed working on also include time done testing, and writing this stage of the report.

Therefore, using the above table with dates of completion, a burndown chart up to this point can be created. The chart in figure 3.5 shows the amount of features predicted remaining to implement versus the amount remaining at this time in the project.

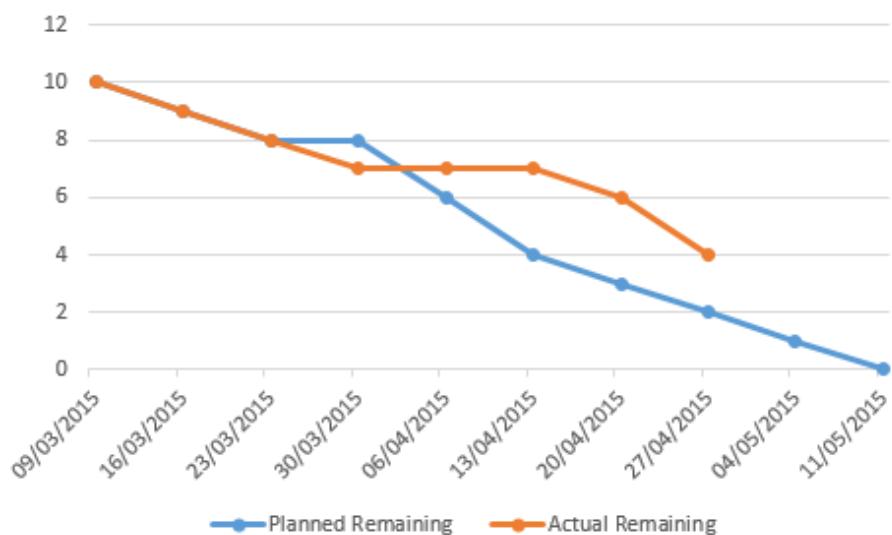


Figure 3.5: Burndown chart of features implemented and tested fully. X axis represents number of features to implement, whilst Y axis shows the date that the features have been implemented by.

Although work appears to be behind schedule here, it should be noted that there is still both time remaining and some features are more than half complete. For strict purposes though and following the FDD manifesto, a feature can not be said to be complete until it is 100% complete.

Analysing the graph further, the main problem can be seen where work was halted on feature five, the creation of manoeuvres on the canvas. Because issues prolonged work here, it meant other features were pushed back until it was decided that they should be completed in order to make a more whole application. Due to moving onto and completing other features first alongside writing this part of the report, considerable time has been made up. The time remaining will now allow

for any completion of features that are most important, and also fixes to others if possible.

In addition to the graph shown, the features can be expressed as feature sets in traditional FDD progress reports. The progress reports using the due dates, and current progress are highlighted in figure 3.6, where the name of the feature set is shown, followed by how many parts the feature set contains, and the percentage. Where the background is green, the feature set is complete, orange means it is currently ongoing, and red means the feature set is behind schedule. It expands on the tasks that need to be done more promptly, so for this project the main issue that will be addressed first is the OLAN to flight path creation.

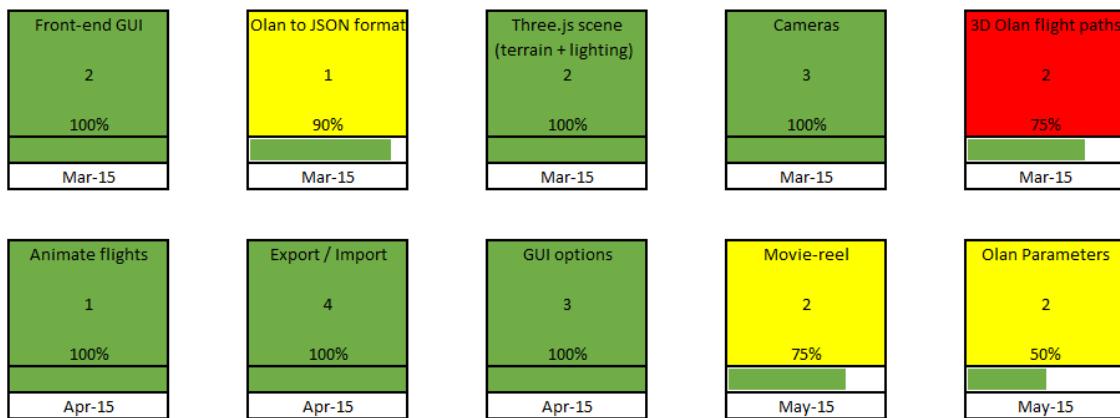


Figure 3.6: Feature progress reports for each of the feature sets implemented up to this point in time. The predicted month of completion is shown at the bottom, and tasks where they are not complete and tested fully are ones without a green background.

3.2.1 After-test fixes and developments

3.3 User testing

Now that the allocated time for implementation and testing has been used, testing to see how the application meets its initial requirements is important. Along with the JSUnit tests and the usability tests that were created, user testing is a means of testing ideal for front-end applications where the usability of the application is paramount. There are a few routes that can be taken to perform user testing, one of which is a questionnaire. Rather than allowing the user to send feedback through suggestions or pros and cons, a quicker and more direct way to check the application fulfills its needs is to provide the user with a set of direct questions, allowing them to state if they agree or disagree. One example could be whether the colour scheme is suitable. By getting the amount agreeing above a certain threshold percentage wise would help show that the colour scheme aspect was correctly applied.

The questionnaire, which can be found in the appendices under appendix D, section ?? provided ten agree or disagree questions that the users had to answer. Upon creating the questionnaire, these were then handed to a pick of 10 users, in technical and none-technical backgrounds to ensure a good spread of opinion. If just technical users were chosen, they might not see the application from the aspect of GUI as much as a standard user, as some users may find it harder to find things on a page. Things such as menus for instance may not be as visible and easy to find for

users who have not seen similar layouts in the past, and it is important that they application has a level of being self-documenting.

The users selected to perform the testing comprised of eight students and two none-students. It was also ensured that half the students selected were not on a computer science related degree to ensure a wider set of technological knowledge. For each test, a brief description of the application was given to the user explaining the purpose and requirements of the project and application, followed by the user trying out different features in the GUI and canvas.

The questionnaires handed out were created using google forms, to keep the process on the internet for easier tracking. Because this form of testing was not planned out originally on the Gantt chart at the start of this project, it was important that the tests were carried out as quickly as possible, so google forms were sent out and asked they could be returned a couple of days later. Indeed all ten questionnaires were returned, each of which all answers had been answered. This was very successful, as it then allowed the results to be analysed properly, and graphs created.

The results of each question can be found in appendix D section ???. One of the results that has been

3.4 Implementation and testing review

On completion of the process of the tests of the final application, a short review of the methods that was used to create the application can now follow. Starting with the pre-coding tasks that had to be completed, Github was initially set up on the machines the application was going to be created on, followed shortly by the set up of Travis and Grunt which was used later for automatic running of tests. Once the repository and branch that would be used to host the application was set up, Sublime text was used to perform the coding, and was used because of the package compatibility to assist with code formatting. In terms of how long it took to prepare the environment for coding, this took less than a day, including the work blog.

This leads up to time management, where the amount of work done in the implementation stage was an important factor. Considering that all features had been started and tested by a week before the stage deadline, progress was kept on top of, and it was ensured that tasks that were taking too long did not affect the rest of the application. An example of some of the issues are ones that came from attempting to create the manoeuvres on the canvas with the OLAN interpretations. The first set back was the interpolation issues, which were fixed by changing the curve inclines. Another setback was the issue with curves not appearing the go in the correct direction, which was again fixed at the end of the implementation.

Testing mechanisms used a wide selection of methods, from usability tests to unit tests ensuring that while the back-end methods performed the features as efficiently as possible, the front end allowed users to interact with these features easily. By using the FDD iterative approach, and producing this section of the report alongside each iteration it has meant that the entire application and documentation has been built in the most stable way. Throughout testing the application, it was ensured the performance levels were checked to be fast enough to run on most devices, primarily using a library called stats.js [?]. This allowed a small framerate counter in the corner of the application, which monitored how many frames or iterations of the animation the application achieved each second. Values between 40 and 60 was the aim, and 60 was the value throughout the most heavy work (having lots of manoeuvres being flown). The framerate counter was displayed

as shown in figure 3.7 with an example of framerate values on different devices in figure 3.8.

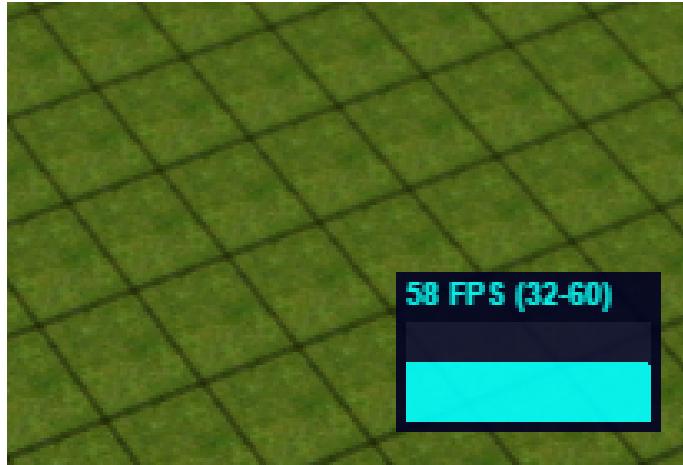


Figure 3.7: Framerate counter shown in bottom right of the application.

In addition to monitoring the stats from the display whilst developing, it was a useful idea to have a more accurate track of how well the application was performing under different browsers and devices. To do this, whilst under the development environment a line of code was inserted into the iteration that got the latest frames per second, and logged this into a csv file. After one minute, the application was then closed, and the average rate was gathered. Each time this was performed on the certain device or browser, it was ensured the same operation occurred to a good accuracy of similarity. The results can be seen in figures 3.9 and 3.8. These tests created a good platform for ensuring a stable application and more efficient code improvements where the framerate may have been dropping.

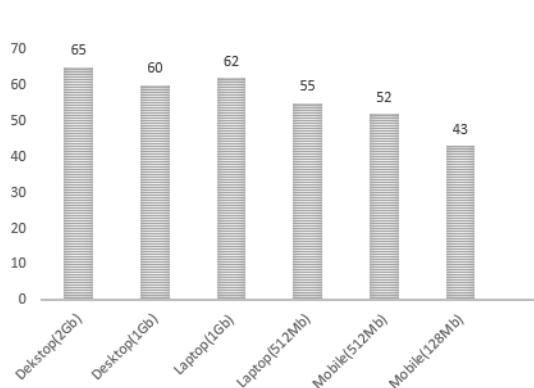


Figure 3.8: Average rate per device(Graphics RAM in brackets)

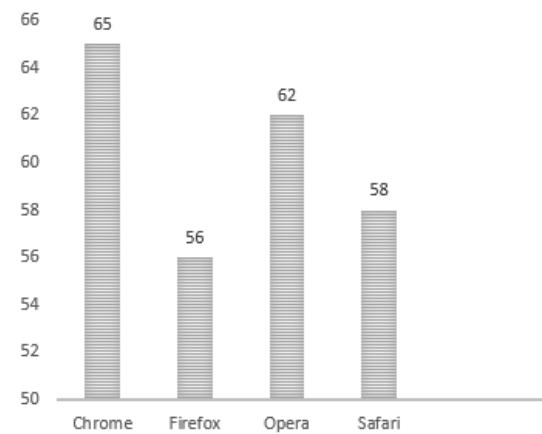


Figure 3.9: Average frame rate per browser (all latest versions of browser on 04/05/2015)

It can be seen that there was one anomaly result, in the desktop with the same graphics memory as the laptop had a lower rate. This could be explained by a number of factors, one such way could be that the processor that would be executing the code may have been slower on the desktop. This was another consideration that ensured optimal coding methods.

There are possible improvements that could have been made to this stage of the project though,

one such would have been sticking to a much stricter order to features, by creating the base logic to produce the manoeuvres first, rather than getting the GUI up and running. This though would have slowed down testing, so there are weigh-ups both ways.

Chapter 4

Evaluation

Upon completion of the application, it is now necessary to evaluate both the application and project as a whole. This section will address a number of topics, in order to grasp an overall effectiveness of the project. For the purposes of highlighting points such as how well requirements were set out, or how well the application fulfilled its purpose, this evaluation has been broken down into the principle stages of the project. At the end, a small self-evaluation will describe what the author of this paper and application has felt was learned through this project in terms of personal and technical achievements.

4.1 Analysis and design decisions

Firstly, answering the question of how successful gathering requirements, the process of this started with looking at the initial project choice and language choice. After giving a brief overview of what the purpose and reasoning for this project, the language choice was an easy decision because WebGL could only run under the JavaScript language. Identifying this, analysing what technologies to use was then found to be much easier.

The amount of research into the background topics was to a good amount that lead up to the discovery of the 15 degree increments idea (though later revised to 45 for interpolation issues)

4.2 Application and testing evaluation

4.3 Project improvements

4.4 Self-Evaluation

Appendices

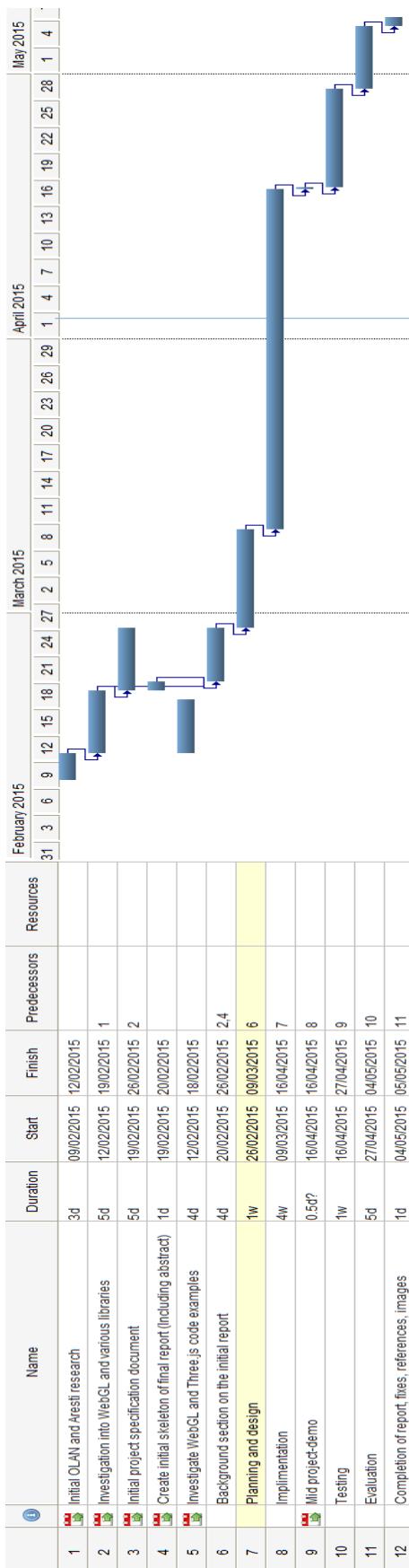
Appendix A

Background research and analysis

1.1 Initial project Gantt chart

See next page (landscape, fits better on an entire page).

Figure A.1: Initial gantt chart, outline times throughout project activities, to be updated as time passes during the project span.



1.2 Initial requirements analysis

Figure A.2: Requirements brief created before a meeting to clarify questions on the initial requirements. The pdf continues onto the next page.

*A WebGL based flight simulator derived from OLAN
(One letter aerobatic notation) inputs.*

Craig Heptinstall (Crh13)

January 21, 2015

This document describes initial requirements and features proposed for the simulator and a broad set of terms detailing technologies and processes used during the project.

Initial general requirements

The listed general requirements are as follows:

1. Provide a web-implemented tool¹ that allows input of the OLAN characters as a string format, alongside possible click functionality.
2. Relate each notation or set of notations to a certain procedural movement² (rotations, movements etc.).
3. Provide a means of linking up these movements in such a way they produce a fluid manoeuvre.
4. Display this using WebGL³. Libraries⁴ to consider that could help with some of the movements:
 - glMatrix- Javascript library for helping with performing actions to matrices- <http://glmatrix.net>
 - ThreeJS- Another Javascript library, good with handling cameras and different views- <http://threejs.org>
5. Allow user to add different effects such as wind, gravity changes and other physics⁵.
6. Add functionalities of different viewpoints(on-board views, side views) to application.
7. Possibility to add function to save (using local storage?) users different sets of manoeuvres?

¹ None-IE due to WebGL capabilities. Will it use a simple JSON file to store notations?

² Must consider parameters in some of the notations, such as the speed of entry into moves, or the angle of the plane.

³ Begin by initially testing simple shapes to move and fly around, then add textures, and plane structure.

⁴ Are libraries ok to use?

⁵ Could be better to implement these last, as it will be easier to test pure functionality of rolls etc first, then figure out natural physics.

Development environments, testing and bug tracking

To develop the project, I have decided on a set of technologies I wish to use:

- To develop on a Github basis- Easier to maintain, links up to build trackers, good room for documentation, code comments etc.
- Use a Travis build server to run tests after each commit- This can be done automatically, provide me some nice statistics, links up to test libraries well.

A WEBGL BASED FLIGHT SIMULATOR DERIVED FROM OLAN (ONE LETTER AEROBATIC NOTATION) INPUTS.

2

- Testing frameworks- I plan to use either libraries such as PhantomJS or Grunt to test my client side code.⁶
- Bug-trackers- For instance inbuilt into Github. Allows me to prioritise higher importance issues. Also helpful for time tracking when adding functionality.

⁶ Need to ensure good de-coupling between data and the shaders etc within WebGL.

Project course specifics

Alongside the functionality of the actual simulator, there are the methods and stages of the project I need to consider:

- Using FDD as a methodology through the process.
 - Using the list given in the first section, make a list of requirements(change these into features).⁷
 - Plan each feature, and design functionality logically and narratively.⁸
 - Implement each feature accordingly, running through coding and testing, then reviewing.
 - Iterate over each feature, until all(or as many as possible) have been completed.
- Document throughout process, any issues, and findings
- Use LaTeX to document⁹

⁷ This could include an overall plan of the project, timing using Gantt charts?

⁸ Sketch-up of plane and angles, and maths behind different transformations.

⁹ Perhaps initial tests or large sets of data can be done by hand and transferred later

1.3 OLAN understandings

Figure A.3: A document created before a second meeting, outlining my current understanding of the OLAN language, and how manoeuvres can be constructed from smaller single-element ones.

Craig Heptinstall (Crh13)

29/01/2015

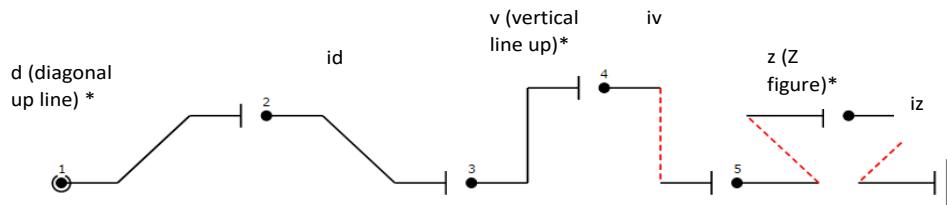
Evaluation of OLAN manoeuvres

This document highlights some of the movements, be that single-element, double-element, loops, double-loops or special and complex moves. Although not all the notated movements presented in the OLAN base figures document are shown here, the most important ones shall be. With each, an explanation of the figure will be displayed, alongside some explanation to the primitive sections of the manoeuvre. This document will serve as a basis of deciding which key line transformations will be required in the simulator, alongside any potential hazardous manoeuvres that may prove difficult.

Note: * Symbols indicate the move can be inverted (using the letter 'l'), ^ indicates the move can be reversed (using the letter 'r').

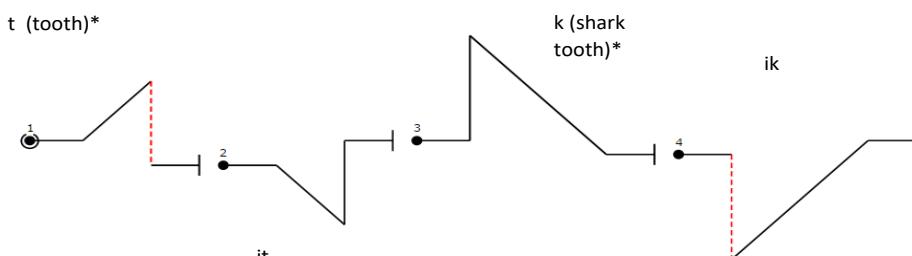
Single-element

Single elements are simply moves that are described as one movement. For example, the first move shown in the figure below is described as a diagonal line up, while move 6 can be simply described as inverted Z figure.



Two-element

A two-element move is a manoeuvre that requires two separate moves to form the entire move. For instance, in the first move, described as a 'tooth' requires firstly a diagonal line up, then an inverted line down. A simple assumption can state that any move in this two-element case can be formed using any of the single-element lines. In move 3, this move known as a 'shark tooth' is comprised of a vertical line up and diagonal line down, though is special in that it can be inverted (by placing an 'i' before the OLAN notation) to mirror the commands.

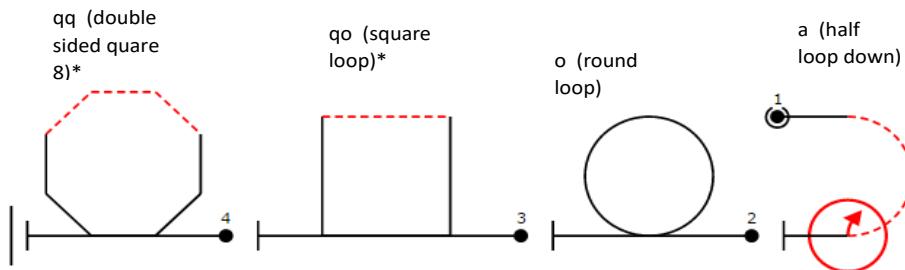


Craig Heptinstall (Crh13)

29/01/2015

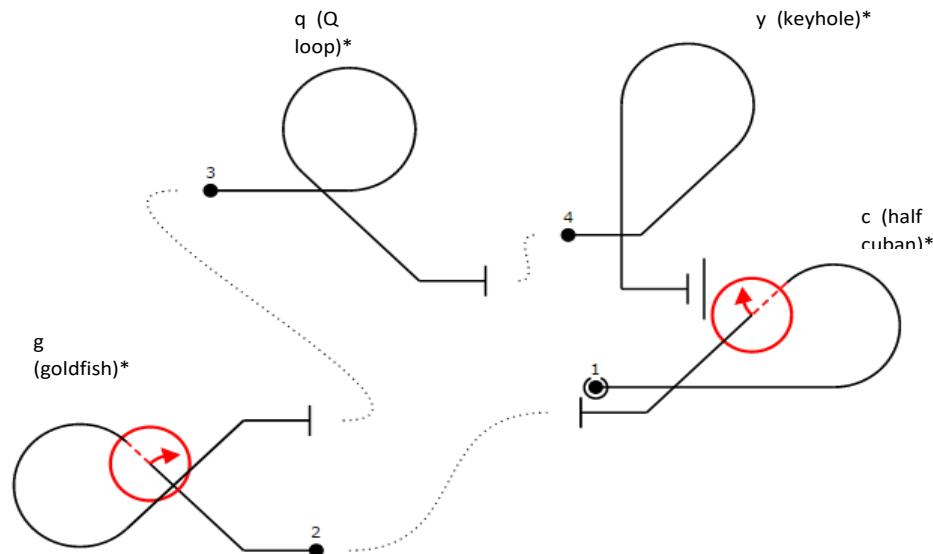
Loops

Loops can be deduced to be single element moves, where the flight consists of one simple move, for instance in move 1, these can be understood to be simply one half loop, containing rolls at the top and bottom of the arc. For each of these loops, they all work by multiples of 45 again, though with examples 4 and 3, rather than a smooth curve, a more square or sudden turn is expected to achieve the desired result. This is one thing that will need to be considered.



Loop-line combinations

A loop-line combination defines moves where both single and two-element lines are combined with loops as shown in the previous section. The first example, which shows a half-Cuban requires a line through to a 5/8 loop (again this is a multiple of 45 degrees) followed by a roll and diagonal line down. Another example shown in move 3, is comprised of a full loop followed by a diagonal line down.

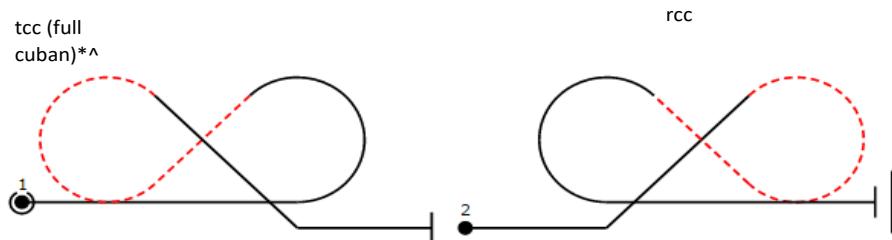


Craig Heptinstall (Crh13)

29/01/2015

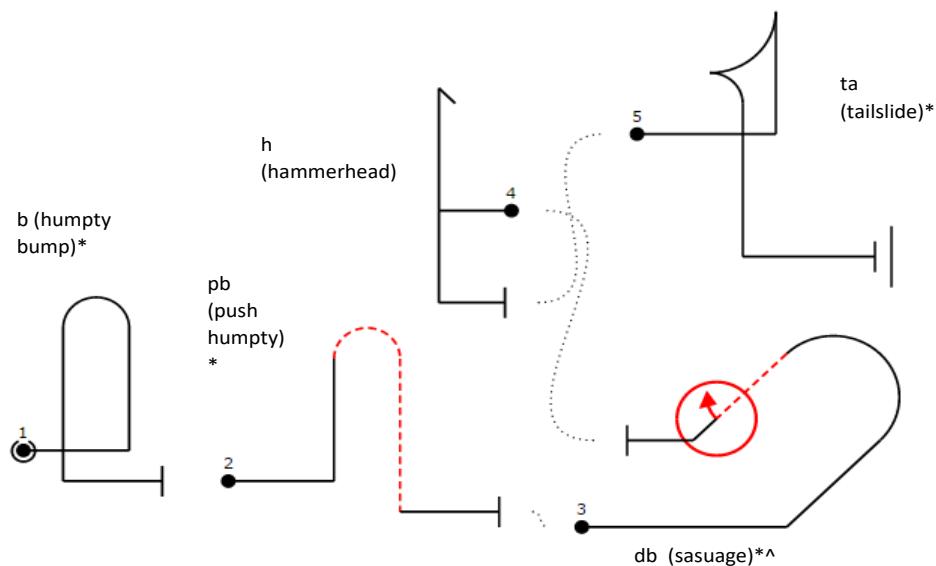
Double-Loops

Double loops are cases where two loops are used after one another to achieve the desired result. The two examples here reflect each other because of the letter 'r' meaning reverse in the second diagram. As you can see, this should be easily achieved by simply swapping over the roll from one loop to the other. Again with these, it is a matter of the angle of turn creating the loop that must be correct to be able to run a diagonal line down to create the next.



Humpty-Dumplings, Hammerheads and Tailslides

A humpty dumpty (1) consists of a bump, followed by a 180 degree turn to come back down vertically. Meanwhile the hammerhead manoeuvre described in move 4 consists of flying straight up, and then rotating on the wing so to fall back down. In diagram 5, the tailslide move also looks a tricky one to implement, as it requires falling back on itself after slowing down. This change of direction in this and in the hammerhead moves could be an issue as simply giving the plane a constant speed in the WebGL program would not work.

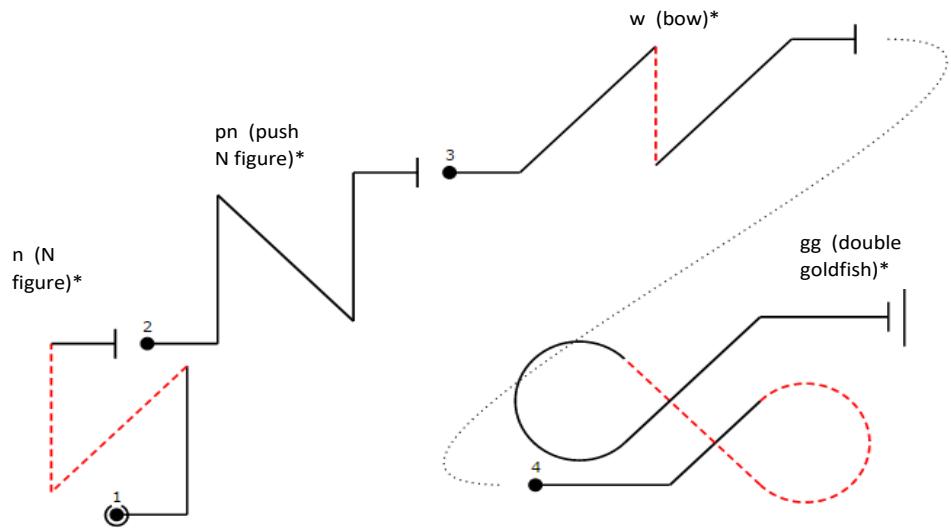


Craig Heptinstall (Crh13)

29/01/2015

Complex 3-Rolling elements

This set of maneuverers contain a total of three different elements, so in diagram one an N figure, which itself is comprised of two moves, followed by a pull back to the diagonal. These moves should not be too difficult to recreate, as each of which should be easily modified previous moves. An example of this is the double goldfish move shown in diagram 4, which is simple a double loop but with more of a tighter loop on either side (this could be defined by size of radius).



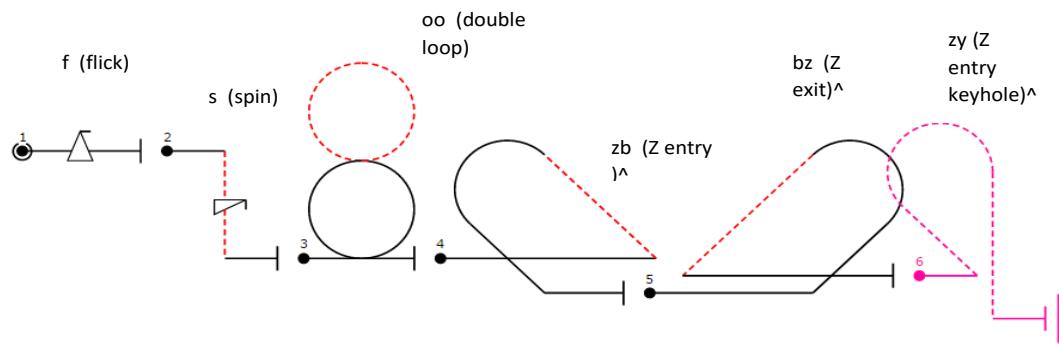
4

Craig Heptinstall (Crh13)

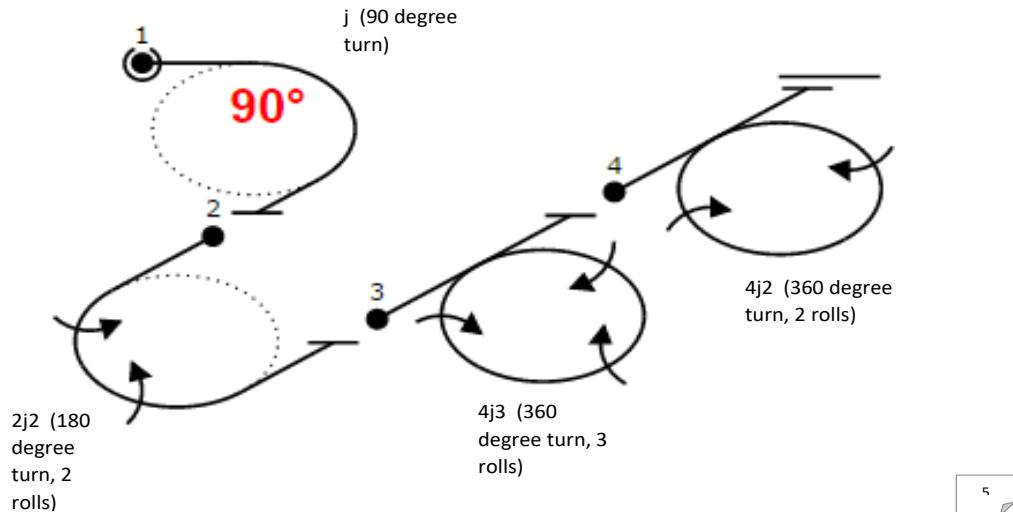
29/01/2015

Oddball and special figures

The moves shown in this sections are ones that are more complex in the way that they require special manoeuvres. Starting with diagrams one and two, they both have a triangle shape placed on the line, indicating a flick roll. This means rolling the plane a total 360 on the horizontal line to return it to its original rotation. Other moves shown below could be recreated again with a use of different single and two element moves. Moves such as diagram 3, would be comprised of two full loops, one inverted, and these loops can be broken down again into 45 degree segments. As the plane leaves the first loop, it should be easy to enter the second already inverted.

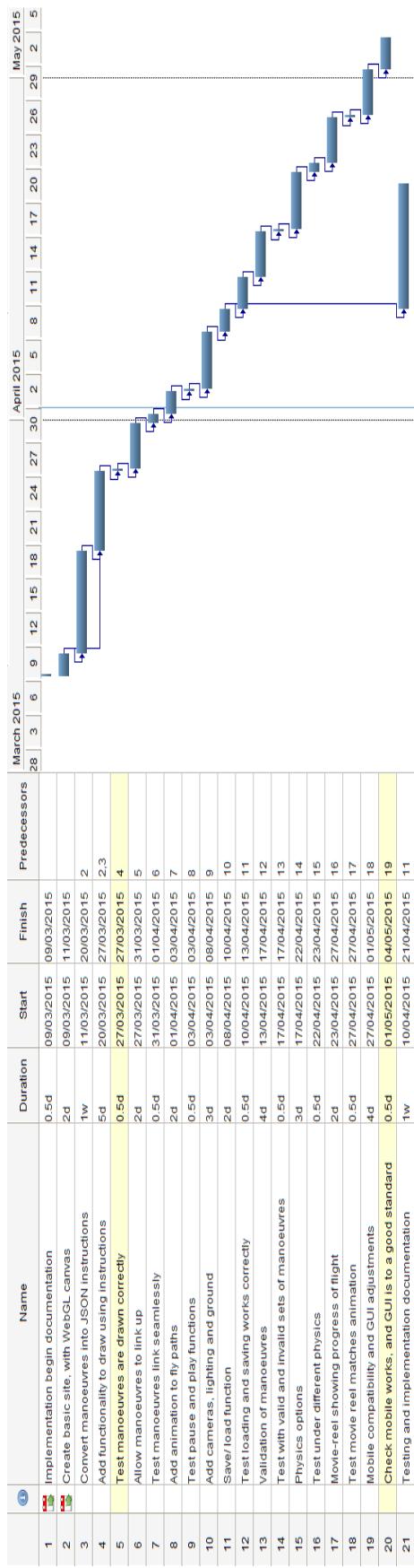


The final set of Aresti moves illustrated here look at turns and rolling turns. The angle in the first move shows a simple 90 degree turn, whilst in the second, because '2' has been placed both sides of the 'j', this signifies 2 90 degree turns, and 2 rolls. This parameter set can be shown in the other two examples, with the last showing 4 90 degree turns (creating a loop), containing 2 rolls.



1.4 Revised FDD Gantt chart

Figure A.4: More detailed Gantt chart, focuses on the implementation and testing strategy of each feature. As you can see, prioritised tasks are the first to be completed.



Appendix B

Third-Party Code and Libraries

If you have made use of any third party code or software libraries, i.e. any code that you have not designed and written yourself, then you must include this appendix.

As has been said in lectures, it is acceptable and likely that you will make use of third-party code and software libraries. The key requirement is that we understand what is your original work and what work is based on that of other people.

Therefore, you need to clearly state what you have used and where the original material can be found. Also, if you have made any changes to the original versions, you must explain what you have changed.

As an example, you might include a definition such as:

Apache POI library The project has been used to read and write Microsoft Excel files (XLS) as part of the interaction with the clients existing system for processing data. Version 3.10-FINAL was used. The library is open source and it is available from the Apache Software Foundation [?]. The library is released using the Apache License [?]. This library was used without modification.

Appendix C

Code samples

3.1 Off-canvas HTML code, showing how menus are hidden

Figure C.1: HTML code showing the page is divided into two, one for the canvas, and one for the menu. The 'off-canvas-list' has sub menus that allow more menus to be hidden to the left.

```
1 <div class="off-canvas-wrap" data-offcanvas>
2   <div class="inner-wrap">
3     <aside class="left-off-canvas-menu">
4       <ul class="off-canvas-list">
5         <li>
6           <label>Option Categories</label>
7         </li>
8         <li class="has-submenu">
9           <li class="has-submenu">
10             <a href="#">OLAN</a>
11           </li>
12           <li class="has-submenu">
13             <a href="#">Physics</a>
14           </li>
15           <li class="has-submenu">
16             <a href="#">Save/Load</a>
17           </li>
18         </li>
19       </ul>
20       <footer id="footer" class="row">
21         </footer>
22     </aside>
23     <div id="container">
24
25       </div>
26       <a class="exit-off-canvas"></a>
27     </div>
28   </div>
```

Figure C.2: JSON code showing a vertical upline defined by JSON instructions.

```

1  {
2      "variant": [
3          "component": [
4              {"_pitch": "NIL",
5               "_roll": "NIL",
6               "_yaw": "NIL",
7               "_length": "10"
8           },
9           {
10              {"_pitch": "POS",
11               "_roll": "NIL",
12               "_yaw": "NIL",
13               "_length": "1"
14           },
15           {
16               {"_pitch": "POS",
17                "_roll": "NIL",
18                "_yaw": "NIL",
19                "_length": "1"
20            },
21            "_olanPrefix": "",
22            "_name": "Vertical up line"
23        },
24        "_olan": "q"
25    },

```

3.2 JSON OLAN instructions example

3.3 RequireJS setup code

Figure C.3: Snippet of code showing how RequireJS starts the application. The RequireJS library is loaded onto the page, the configuration telling the library the base of the application JavaScript files is set, and then is told to start by running the main.js file.

```

1 <script>
2     require.config({
3         baseUrl: 'js',
4     });
5     require(["main"]);
6 </script>

```

3.4 Converting JSON representations into vectors

Figure C.4

```

1 for (m in values) {
2
3     var components = values[m].components;
4
5     for (var i = 0; i < components.length; i++) {
6         var component = components[i];
7         var prevVector = new THREE.Vector3(0, 0, 0);
8
9         if (linePoints.length > 0)
10            prevVector.copy(linePoints[linePoints.length - 1]);
11
12         if (component.PITCH == 0 && component.YAW == 0 && component.ROLL == 0)
13             {
14                 prevVector.setZ(prevVector.z + (component.LENGTH * 10));
15                 linePoints.push(prevVector);
16             } else {
17                 var axis = new THREE.Vector3(1, 0, 0);
18                 var angle = Math.PI / 180 * angleDiv * -component.PITCH;
19                 prevVector.applyAxisAngle(axis, angle);
20
21                 var axis = new THREE.Vector3(0, 1, 0);
22                 var angle = Math.PI / 180 * angleDiv * component.YAW;
23                 prevVector.applyAxisAngle(axis, angle);
24
25                 prevVector.setZ(prevVector.z + (component.LENGTH * 10));
26                 linePoints.push(prevVector);
27             }
28         var startVector = new THREE.Vector3(0, 0, 0);
29         startVector.copy(prevVector);
30         linePoints.push(startVector);
31     }

```

3.5 Using local storage to export scenarios

Figure C.5

```

1 exportToLocalStorage: function(manoeuvres) {
2     if (checkLocalStoragePermitted()) {
3         if (!localStorage.getItem("autoLoad"))
4             return;
5         // Store
6         localStorage.setItem("manoeuvres", manoeuvres);
7     }
8 }

```

Appendix D

Application tests

4.1 Usability testing for the GUI front-end

Test	Pred. Result	Pass/ Fail	Comment
Open menu	Menu is displayed on the left, canvas moves right.	Pass	
Close menu	Menu hides to the left, canvas is pulled back in to fill page.	Pass	
Resize window to mobile size	Menu changes to mobile view, canvas still fills page.	Pass	
Open mobile menu	Menu displaying OLAN input, about and help pages listed.	Pass	
Open help page	Overlay appears from above, covers canvas with white box and text.	Pass	
Open about page	Overlay appears from above, covers canvas with white box and text.	Pass	
Close about page	Box fades away.	Pass	
Open sub-menu	Menu pulls in from left in side to cover current menu. Shows options of certain menu chosen.	Pass	
Click back on sub-menu	Sub-menu slides to left and reveals original top level menu.	Pass	

Table D.1: Usability test table for testing the GUI.

4.2 Test Unit code for converting JSON to manoeuvre instruction object

Figure D.1

4.3 Test Unit code for checking lights and ground are created successfully

Figure D.2

4.4 Usability testing for cameras

Test	Pred. Result	Pass/ Fail	Comment
Test drag mouse to rotate around canvas	Camera should rotate around the scene.	Fail	Changed after fail, due to camera rotation around the point(0,0,0). Changed now so that it rotates around current camera location.
Scroll zoom in	Camera should focus in on the center of the canvas.	Pass	
Scroll zoom out	Camera should see more of the canvas as mouse scrolls out.	Pass	
Left keyboard key to move left	Camera moves 10px to the left.	Pass	
Right keyboard key to move right	Camera moves 10px to the right.	Pass	
Up keyboard key to move forward	Camera moves 10px forward its Z axis.	Pass	
Down keyboard key to move backwards	Camera moves 10px backwards its Z axis.	Pass	
Switch to onboard camera view	Camera changes to 0,0,0 looking ahead.	Pass	

Table D.2

4.5 Test Unit code for animation controller getters and setters

Figure D.3

4.6 Test Unit code for checking import and export to JSON/ local storage functionality

Figure D.4

4.7 Usability testing for GUI options

4.8 Usability testing providing parameters between OLAN

Test	Pred. Result	Pass/ Fail	Comment
Type OLAN into input box to create figures	Loop should appear when 'o' is typed in, and movie-reel should show manouvre at bottom.		
Use dropdown in OLAN options to add a figure	Loop should be added to canvas when chosen from dropdown.	Pass	
Increase and decrease scale	Figures on canvas should become bigger or smaller depending on scale.	Pass	
Increase and decrease extrusion segments	Curves should become less and more smooth depending on extrusion.	Pass	
Increase and decrease radius	More sides to the figure should appear the higher the radius.	Pass	
Increase animation speed	Speed of flight should increase depending on slider of speed.		
Turn switch on for auto repeat of animation	When on, when animating, should loop from beginning to end endlessly.	Pass	
Switch auto save on and close application, then re-open	When on, if application is closed and re-opened then routines should be added automatically. Vice versa.	Pass	
Export to JSON using button	JSON file begins to download on click.	Pass	
Import JSON file using file chooser	Routine should load up when selecting a file.	Pass	

Table D.3

Test	Pred. Result	Pass/ Fail	Comment
Enter empty parameters '()'	No change to start location of next manoeuvre should occur.	Pass	
Empty correct format parameters '(10,10)'	Should move start of next manoeuvre 10 pixels up, 10 to the left.	Pass	
Enter parameters without space between OLAN	The manouvre nor parameters will be placed on the canvas.	Pass	
Enter parameters with negative values '(-10,-10)'	Should move start of next manoeuvre 10 pixels down, 10 to the right.	Pass	
Enter parameters on a manoeuvre "o""	Loop should be drawn with three times the length exit travel.	Fail	Parameter had np effect to the loops exit, and prevented the manouvre from being drawn.

Table D.4

Annotated Bibliography

- [1] Michael Golan, “Olan Welcome & Support page.” <http://web.archive.org/web/20060927153819/http://www.aerobatics.org.il/olan/welcome.php>, 2006.
- [2] Ringo Massa, “OpenAero figures reference guide.” <http://www.openaero.net/language.html>, 2012.
- [3] Ringo Massa, “OpenAero main application.” <http://www.openaero.net>, 2012.
- [4] The British Aerobic Association Ltd, “Aresti Catalogue introduction.” <https://www.aerobatics.org.uk/aresti>, 2015.
- [5] B. Wegman pp. 1–9.
- [6] Brandon Jones, “glMatrix vector library.” <http://glmatrix.net>, 2011.
- [7] Ricardo Cabello, “The three.js library documentation.” <http://threejs.org/docs/>, 2010.
- [8] Mark Pilgrim, “The past, present and future of local storage for web applications.” <http://diveintohtml5.info/storage.html>, 2011.
- [9] Scott W. Ambler, “Feature Driven Development (FDD) and Agile Modeling.” <http://www.agilemodeling.com/essays/fdd.htm>, 2005.
- [10] Carl Danley, “The Observer Pattern.” <https://carldanley.com/js-observer-pattern/>, 2014.
- [11] Martin Fowler, “LazyInitialization.” <http://martinfowler.com/bliki/LazyInitialization.html>, 2015.
- [12] Addy Osmani, “Learning JavaScript Design Patterns.” <http://addyosmani.com/resources/essentialjsdesignpatterns/book/>, 2014.
- [13] Aaron Whyte, Bob Jervis, Dan Pupius, Erik Arvidsson, Fritz Schneider, Robby Walker, “Google JavaScript Style Guide.” <https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>, 2012.
- [14] ZURB Inc., “Foundation.” <http://foundation.zurb.com>, 1998.
- [15] Github inc, “Github.” <https://www.github.com>, 2015.
- [16] Githib inc, “Websites for you and your projects..” <https://pages.github.com>, 2015.

- [17] Travis CI, “Easily sync your GitHub projects with Travis CI..” <https://travis-ci.org>, 2015.
- [18] Gregg Van Hove, “Behavior-Driven JavaScript.” <http://jasmine.github.io>, 2008.
- [19] Jeff Williams, “@use JSDoc.” <http://usejsdoc.org>, 2011.
- [20] Hampton Catlin, Natalie Weizenbaum, Chris Eppstein, “Sass: Syntactically Awesome Style Sheets.” <http://sass-lang.com>, 2006.
- [21] Ethan Lai, “Koala - a gui application for LESS, Sass, Compass and coffeescript.” <http://koala-app.com>, 2013.
- [22] EasyPHP, “EasyPHP — Install a local WAMP server.” <http://www.easyphp.org/introduction.php>, 2000.
- [23] Michael Golan, “Olan Language Basics.” <http://web.archive.org/web/20060927153819/http://www.aerobatics.org.il/olan/language.php>, 2006.
- [24] Twitter, “Bootstrap.” <http://getbootstrap.com>, 2012.