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 apparant and in an orgaised 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 . . . 12
  - 2.2.4 Naming conventions . . . 14
- 2.3 User Interface . . . 15
  - 2.3.1 GUI Use-case . . . 15
  - 2.3.2 CSS and wireframe design . . . 15
- 2.4 Planned use of services . . . 15

3 Implementation **16**

4 Testing **17**
- 4.1 Overall Approach to Testing . . . 17
- 4.2 Automated Testing . . . 17
  - 4.2.1 Unit Tests . . . 17
  - 4.2.2 User Interface Testing . . . 17
  - 4.2.3 Stress Testing . . . 17
  - 4.2.4 Other types of testing . . . 17
- 4.3 Integration Testing . . . 17
- 4.4 User Testing . . . 17

5 Evaluation **18**

Appendices **19**

A Background research and analysis **20**
- 1.1 Initial project Gantt chart . . . 20
- 1.2 Inital requirements analysis . . . 22
- 1.3 OLAN understandings . . . 24
- 1.4 Revised FDD Gantt chart . . . 30

B Third-Party Code and Libraries **32**

C Code samples **33**
- 3.1 Random Number Generator . . . 33

Annotated Bibliography **36**

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. 11

A.1 Initial gantt chart, outline times throughout project activities, to be updated as time passes during the project span. 21
A.2 Requirements brief created before a meeting to clarify questions on the initial requirements. The pdf continues onto the next page. 22
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. 25
A.4 More detailed Gantt chart, focuses on the implimentation and testing stratedgy of each feature. As you can see, prioritised tasks are the first to be completed. 31

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

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 [4] 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' [6]. 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 [7] 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 [9] 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 [10], 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 [6] 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 1.1 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.

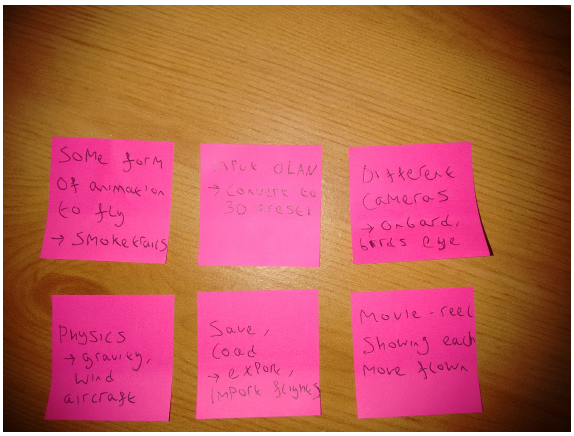


Figure 1.1: Image of my initial user stories after my first meeting, short and concise requirements.

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 [1]

- ThreeJS- Another JavaScript library, good with handling cameras and different views [5]

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 1.2 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 manoeuvres. 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-Dumpties, Hammerheads and Tailslides- Each of these manoeuvres represent specific shapes, such as a 'humpty dumpty' 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 1.3 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 none-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

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

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.

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 [5] 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 [2] 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 me 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 [8] 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 1.4 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. Becuase 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 becuase 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 disaplaying 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 relevent 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.

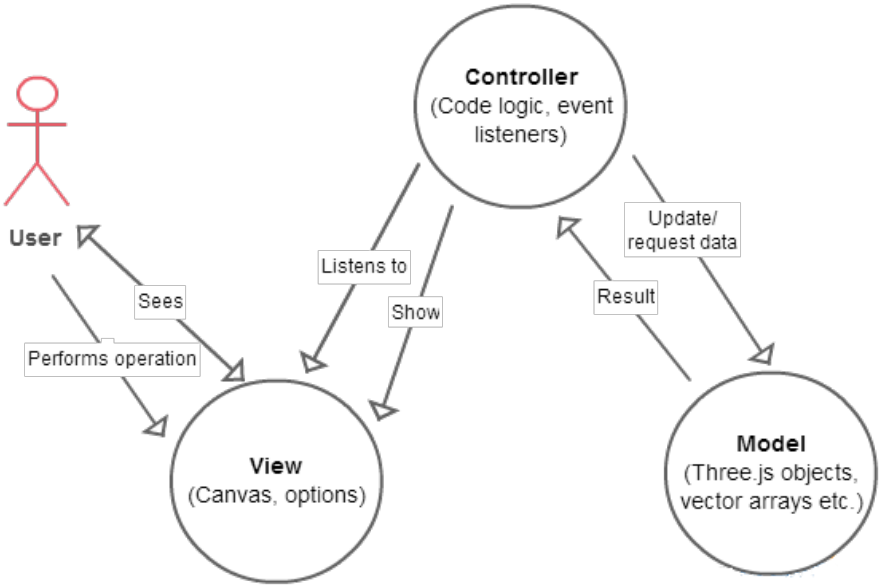


Figure 2.1: Model View Controller diagram for my proposed application. Shows the planned interactions between different subjects within my application.

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 implimentation 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 relevent 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 archetecture, a standalone file will be responsible for listening to any options or menu changes, whilst another JavaScript file will be repsonsible for listening to the canvas events such as rotation or zoom.

The next design pattern I plan on implementing into my application relates again to the MVC archetecture I will be using. The builder pattern will allow me to append and edit any HTML on

the page(such as options, checkboxes, loading content back into the OLAN input box) easily with data retreieved from the model. JQuery again should help to provide a means of changing content, becuase 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 nessecary 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 initialised. One motivation for the need for this pattern in my program is becuase 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.

Although the previous design patterns have been chosen, others were considered when planning the application but were not suitible or better options were availible. 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 repition of functions. In my proposed application, I thoguht about the possibility of using a Mixin style architecture to allow functions that would draw both the manouevres on the canvas, and the manoeuvres 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.

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 archetecture 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 accessable 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 seperate 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 appearence from back-end. Also including the possible movie reel live animatiion.
- Canvas controls- Allowing the user to move along the canvas, and zoom.

Now I have a stable list of categoried features, I am able to create a diagram shown in figure 2.2 to represent what modulated layout my applicatioon will use. Becuase 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 peices 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 maintenence and readability. The following descriptions have been kept brief, and key connections within the diagram discussed.

The first module propsed 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 nessecary calls to set up the listeners for the cameras, canvas and options. The most importat 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 chaning animatiton 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 webpage 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, retreving input from the OLAN box, getting values of any checkboxes, and also setting any values. This module will be using the JQuery library as well as the Handlbars library as the primary way to communicate with the front-end, by diectly referencing ID's of divs on the webpage.

Two other controllers that should be mentioned are the camera dna canvas modules. Both modules will be resposible for setting up their elements when the applicaton starts up (including the different sets of cameras, locations, lighting and ground effects) and for listening to evens 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. Becuase 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 relativly 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, espeically the OLAN instructions for construction of each manoeuvre is organised effiecently and be as accessable 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:

```
{
    "catalogue": {
        "manoeuvre": [{
            "variant": [{
                "component": [{
                    "_pitch": "NIL",
                    "_roll": "NIL",
                    "_yaw": "NIL",
                    "_length": "1"
                }, {
                    "_pitch": "POS",
                    "_roll": "NIL",
                    "_yaw": "NIL",
                    "_length": "1"
                }],
                "_olanPrefix": "",
                "_name": "example cuvre up"
            }],
            "_olan": "u"
        }]
    }
}
```

Listing 2.2: A JSON means of holding break downs of manouvres with each one holding information on different varients 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 labelled 'catalogue' will hold the entire list of OLAN notations (without the prefixes and postfixes for reversing or inversing moves), with each holding another array of varients. Within each varient, 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 noticable

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 precompile vectors from OLAN entry before saving could appear as such:

```
{
  "Manouuvres":{[
    "Move": {
      "Vectors"{[
        {
          "Vector" : "2, 1, 0"
          "Rotation" : "0"
        },
        {
          "Vector" : "5, 1, 0"
          "Rotation" : "0"
        }
      ]},
      "OLAN": "u"
    }
  ]}
}
```

Listing 2.3: A JSON means of holding break downs of manouvres with each one holding information on different varients 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:

```
{
  "OLAN" : "o id b"
}
```

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

Once other predecessing features have been created and time is availible to add the save/load feature, then both ooptions will be considered again. For the time being, it would be easier to choose the second option becuase 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 namning conventions. It is important that variables, methods and moudles are named accordingly to thier function, and so that they are easily readable. For the puirpose of my application, I will be ensuring to use the Google reccomendations for naming conventions in Javascript. The guide, which can be found **here**, 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

2.3.1 GUI Use-case

2.3.2 CSS and wireframe design

2.4 Planned use of services

Chapter 3

Implementation

The implementation should look at any issues you encountered as you tried to implement your design. During the work, you might have found that elements of your design were unnecessary or overly complex; perhaps third party libraries were available that simplified some of the functions that you intended to implement. If things were easier in some areas, then how did you adapt your project to take account of your findings?It is more likely that things were more complex than you first thought. In particular, were there any problems or difficulties that you found during implementation that you had to address? Did such problems simply delay you or were they more significant? You can conclude this section by reviewing the end of the implementation stage against the planned requirements.

Chapter 4

Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.Have you tested your system on real users? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.The following sections indicate some areas you might include. Other sections may be more appropriate to your project.

4.1 Overall Approach to Testing

4.2 Automated Testing

4.2.1 Unit Tests

4.2.2 User Interface Testing

4.2.3 Stress Testing

4.2.4 Other types of testing

4.3 Integration Testing

4.4 User Testing

Chapter 5

Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree. There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved. Review the discussion on the Evaluation section from the lectures. A recording is available on Blackboard.

Appendices

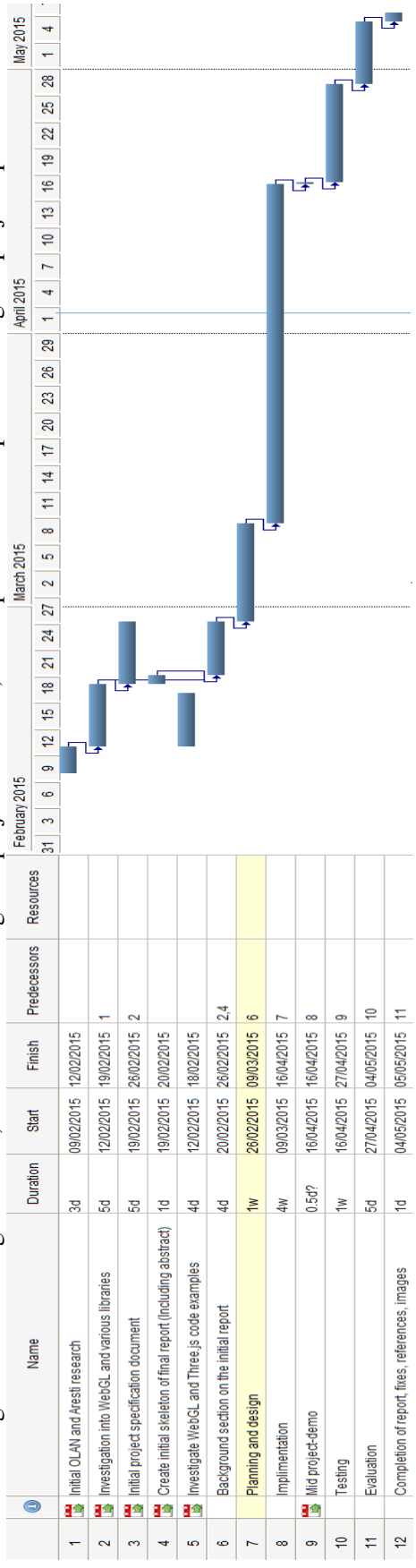
Appendix A

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.



	Name	Duration	Start	Finish	Predeecessors	Resources
1	Initial OLAN and Aresti research	3d	09/02/2015	12/02/2015		
2	Investigation into WebGL and various libraries	5d	12/02/2015	19/02/2015	1	
3	Initial project specification document	5d	19/02/2015	26/02/2015	2	
4	Create initial skeleton of final report (Including abstract)	1d	19/02/2015	20/02/2015		
5	Investigate WebGL and Three.js code examples	4d	12/02/2015	18/02/2015		
6	Background section on the initial report	4d	20/02/2015	26/02/2015	2,4	
7	Planning and design	1w	26/02/2015	09/03/2015	6	
8	Implimentation	4w	09/03/2015	16/04/2015	7	
9	Mid project-demo	0.5d?	16/04/2015	16/04/2015	8	
10	Testing	1w	16/04/2015	27/04/2015	9	
11	Evaluation	5d	27/04/2015	04/05/2015	10	
12	Completion of report, fixes, references, images	1d	04/05/2015	05/05/2015	11	

1.2 Inital 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 brad 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 tool[1] 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 movement[2] (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 WebGL[3]. Libraries[4] 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 physics[5].
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 strcture.

[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.

- 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 document[9]

[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 undertsanding of the OLAN language, and how manoeuvres can be constructed from smaller single-element ones.

Craig Heptinstall (Crh13) 29/01/2015

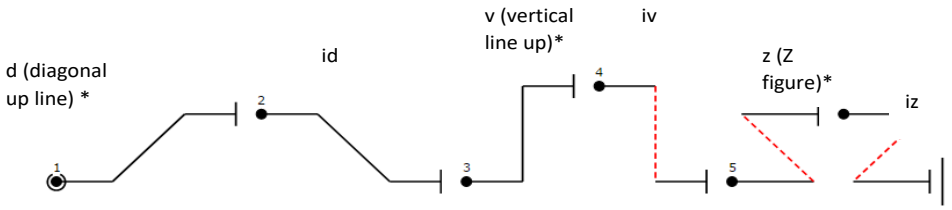
Evaluation of OLAN manoeuvres

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 'I'), ^ indicates the move can be reversed (using the letter 'r').

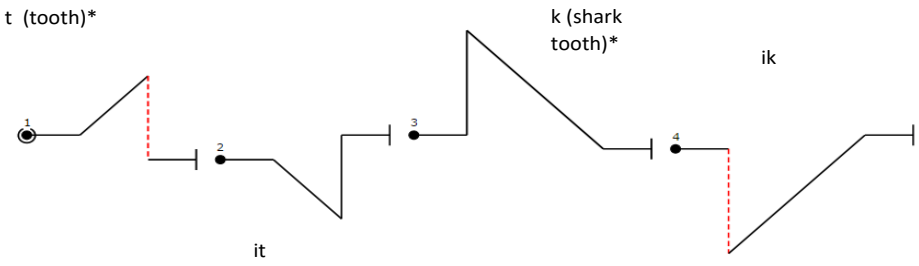
Single-element

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 manoeuver 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 know 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.

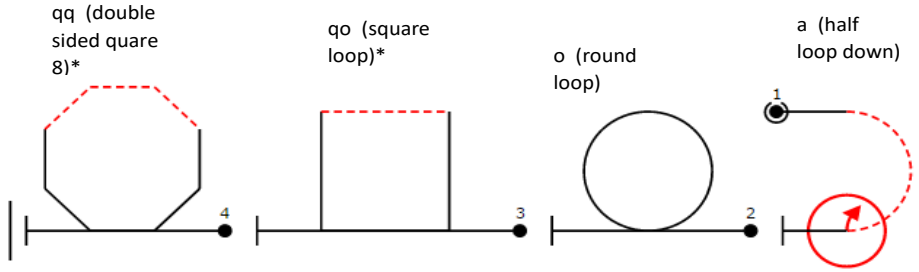


1

Craig Heptinstall (Crh13) 29/01/2015

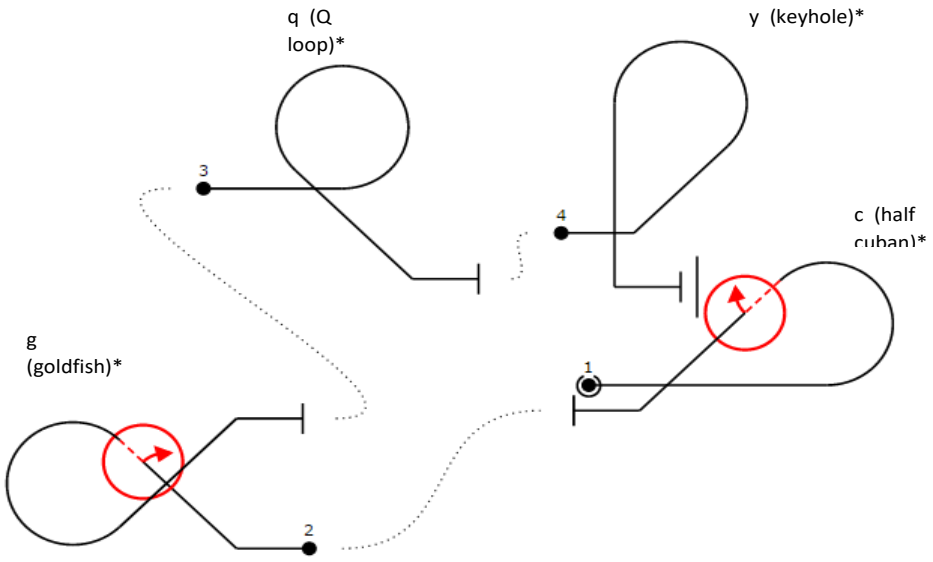
Loops

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

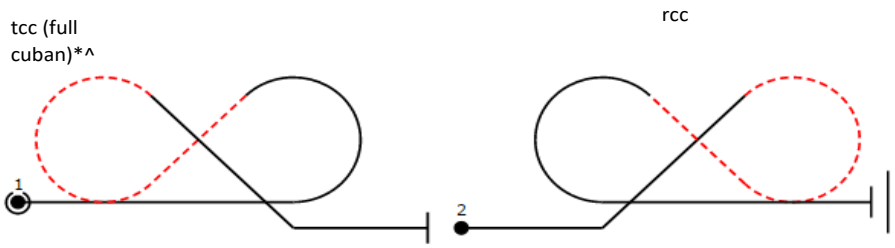
A loop line combination defines moves where both single and two-element lines are combined with loops as shown in the previous section. The first example, which shows a half-Cuban requires a line through to a 5/8 loop (again this is a multiple of 45 degrees) followed by a roll and diagonal line down.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



Craig Heptinstall (Crh13) 29/01/2015

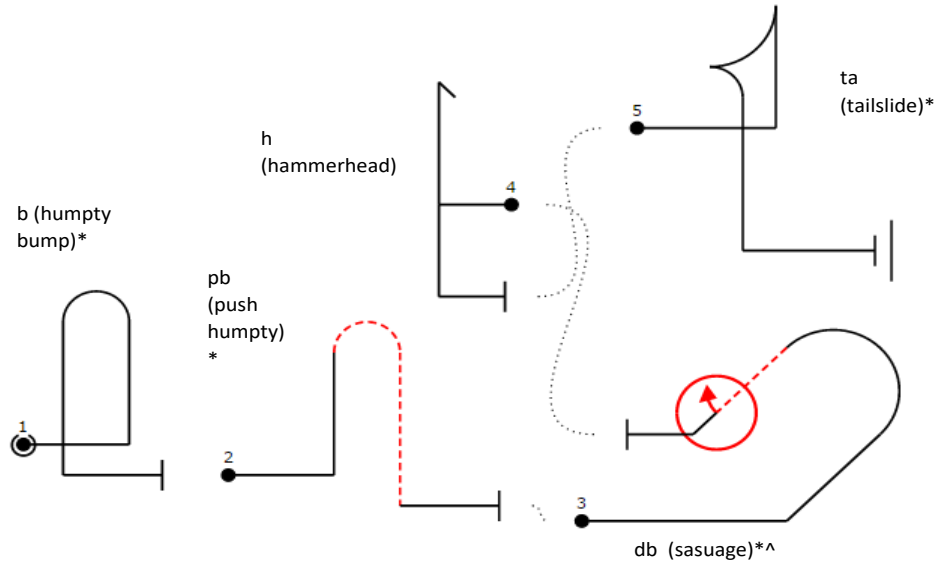
Double-Loops

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-Dumpties, Hammerheads and Tailslides

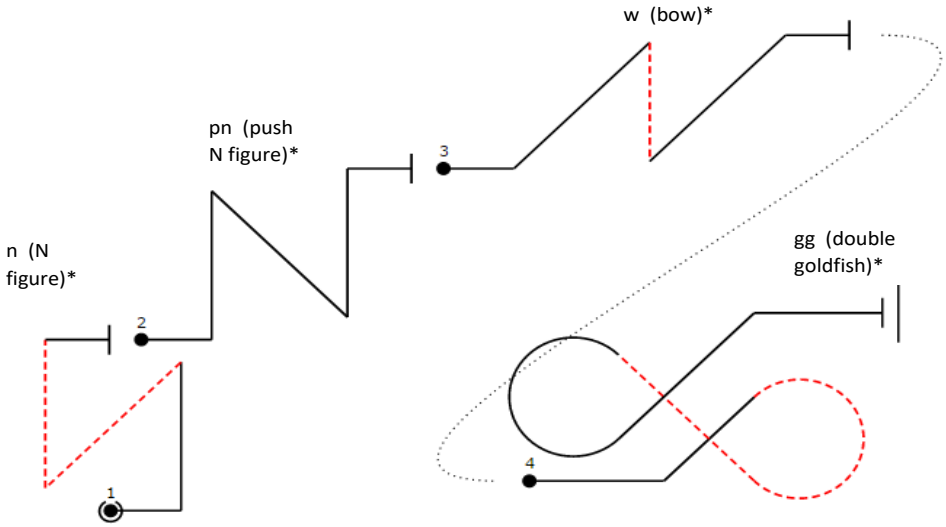
A humpty dumpty (1) consists of a bump, followed by a 180 degree turn to come back down vertically. Meanwhile the hammerhead manoeuvre described in move 4 consists of flying straight up, and then rotating on the wing so to fall back down. In diagram 5, the tailslide move also looks a tricky one to implement, 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

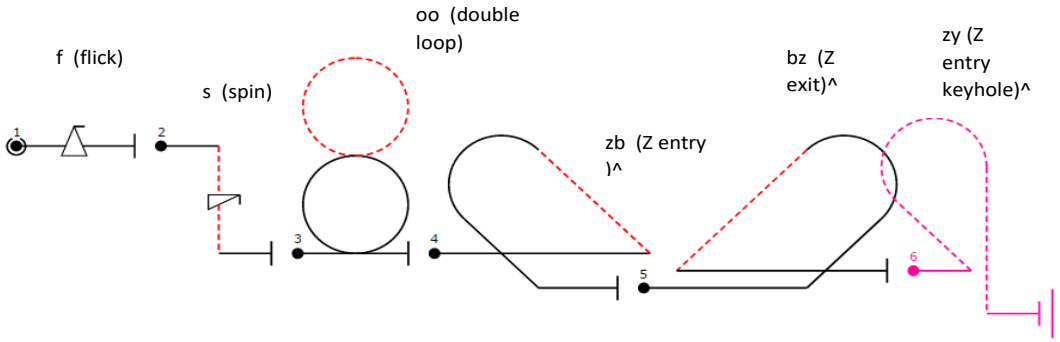
This set of maneuverers contain a total of three different elements, so in diagram one an N figure, which itself is comprised of two moves, followed by a pull back to the diagonal. These moves should not be too difficult to recreate, as each of which should be easily modified previous moves. An example 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).



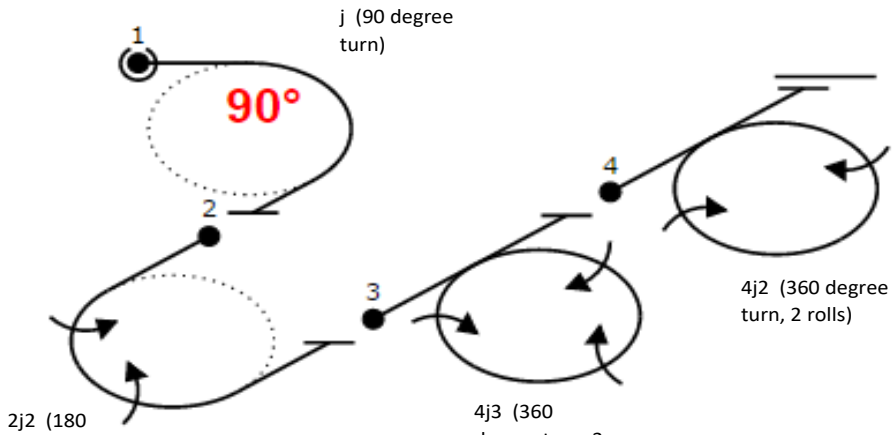
Craig Heptinstall (Crh13) 29/01/2015

Oddball and special figures

The moves shown in this sections are ones that are more complex is the way that they require special manoeuvres. Starting with diagrams one and two, they both have a triangle shape places 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 places 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 implimentation and testing stratedgy of each feature. As you can see, prioritised tasks are the first to be completed.

	Name	Duration	Start	Finish	Predecessors
1	Implementation begin documentation	0.5d	09/03/2015	09/03/2015	
2	Create basic site, with WebGL canvas	2d	09/03/2015	11/03/2015	
3	Convert manoeuvres into JSON instructions	1w	11/03/2015	20/03/2015	2
4	Add functionality to draw using instructions	5d	20/03/2015	27/03/2015	2,3
5	Test manoeuvres are drawn correctly	0.5d	27/03/2015	27/03/2015	4
6	Allow manoeuvres to link up	2d	27/03/2015	31/03/2015	5
7	Test manoeuvres link seamlessly	0.5d	31/03/2015	01/04/2015	6
8	Add animation to fly paths	2d	01/04/2015	03/04/2015	7
9	Test pause and play functions	0.5d	03/04/2015	03/04/2015	8
10	Add cameras, lighting and ground	3d	03/04/2015	08/04/2015	9
11	Save/load function	2d	08/04/2015	10/04/2015	10
12	Test loading and saving works correctly	0.5d	10/04/2015	13/04/2015	11
13	Validation of manoeuvres	4d	13/04/2015	17/04/2015	12
14	Test with valid and invalid sets of manoeuvres	0.5d	17/04/2015	17/04/2015	13
15	Physics options	3d	17/04/2015	22/04/2015	14
16	Test under different physics	0.5d	22/04/2015	23/04/2015	15
17	Movie-reel showing progress of flight	2d	23/04/2015	27/04/2015	16
18	Test movie reel matches animation	0.5d	27/04/2015	27/04/2015	17
19	Mobile compatibility and GUI adjustments	4d	27/04/2015	01/05/2015	18
20	Check mobile works, and GUI is to a good standard	0.5d	01/05/2015	04/05/2015	19
21	Testing and implementation documentation	1w	10/04/2015	21/04/2015	11

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 Random Number Generator

The Bayes Durham Shuffle ensures that the psuedo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs [**?**].

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0 - EPS)

double ran2(long *idum)
{
  /*-------------------------------------------------*/
  /* Minimum Standard Random Number Generator        */
  /* Taken from Numerical recipies in C              */
  /* Based on Park and Miller with Bays Durham Shuffle */
  /* Coupled Schrage methods for extra periodicity   */
  /* Always call with negative number to initialise  */
  /*-------------------------------------------------*/

  int j;
  long k;
  static long idum2=123456789;
```

```
static long iy=0;
static long iv[NTAB];
double temp;

if (*idum <=0)
{
  if (-(*idum) < 1)
  {
    *idum = 1;
  }else
  {
    *idum = -(*idum);
  }
  idum2=(*idum);
  for (j=NTAB+7;j>=0;j--)
  {
    k = (*idum)/IQ1;
    *idum = IA1 *(*idum-k*IQ1) - IR1*k;
    if (*idum < 0)
    {
      *idum += IM1;
    }
    if (j < NTAB)
    {
      iv[j] = *idum;
    }
  }
  iy = iv[0];
}
k = (*idum)/IQ1;
*idum = IA1*(*idum-k*IQ1) - IR1*k;
if (*idum < 0)
{
  *idum += IM1;
}
k = (idum2)/IQ2;
idum2 = IA2*(idum2-k*IQ2) - IR2*k;
if (idum2 < 0)
{
  idum2 += IM2;
}
j = iy/NDIV;
iy=iv[j] - idum2;
iv[j] = *idum;
if (iy < 1)
{
  iy += IMM1;
}
```

```
  if ((temp=AM*iy) > RNMX)
  {
    return RNMX;
  }else
  {
    return temp;
  }
}
```

Annotated Bibliography

[1] Brandon Jones, “glMatrix vector library,” http://glmatrix.net, 2011.

Javascript Matrix and Vector library for High Performance WebGL apps

[2] Mark Pilgrim, “The past, present and future of local storage for web applications,” http://diveintohtml5.info/storage.html, 2011.

A thoughough guide to Javascript local storage, including code examples.

[3] Michael Golan, “Olan Language Basics,” http://web.archive.org/web/20060927153819/http://www.aerobatics.org.il/olan/language.php, 2006.

A web archive of the basics and list of OLAN notations from its original creator- This page was taken off-line in 2012.

[4] ——, “Olan Welcome & Support page,” http://web.archive.org/web/20060927153819/http://www.aerobatics.org.il/olan/welcome.php, 2006.

A web archive of the original idea of OLAN, from its original creator- This page was taken off-line in 2012.

[5] Ricardo Cabello, “The three.js library documentation,” http://threejs.org/docs/, 2010.

Documentation for the three.js library, including samples, manual, and refrences to components and objects in the library.

[6] Ringo Massa, “OpenAero figures reference guide,” http://www.openaero.net/language.html, 2012.

Guide to the OpenAero language, and help with what each manoeuvre means, including parameters.

[7] ——, “OpenAero main application,” http://www.openaero.net, 2012.

The application allowing input of OLAN to produce Aresti diagrams. Currently open source and licensed under GPL V3

[8] Scott W. Ambler, “Feature Driven Development (FDD) and Agile Modeling,” http://www.agilemodeling.com/essays/fdd.htm, 2005.

A guide to the Feature driven development life cycle. Explaings in detail each stage, with annotated diagrams to show how to perform each part.

[9] The British Aerobatic Association Ltd, "Aresti Catalogue introduction," https://www.aerobatics.org.uk/aresti, 2015.

Background information on Aresti from the aerobatics society.

[10] B. Wegman, pp. 1–9.

Detailed description on Aresti figures, broken down to key component explanation