B.Eng. Dissertation

3D PLOTTING OF AIRCRAFT ATTITUDE

by
MUHAMMAD OMER IQBAL

National University of Singapore
2013/2014

Project ID: H100350
Project Supervisor: Dr. Colin Tan
Deliverables:
Report: 1 Volume

ABSTRACT

Air accident investigations usually involve reading and interpreting large volumes of flight data. Visualising this data, especially in a geographic context makes its analysis easier. Therefore the Air Accident Investigation Board (AAIB) needed a tool that would render a 3 dimensional model of flight data from their datasets. The data includes measurements like latitude, longtitude, altitude, heading, roll, thrust etc over time. To meet this criteria I designed a Clojure based application that takes a dataset in csv format and renders aircraft attitude over time on a map. The application generates a KML (Keyhole Markup Language) file which can be opened on Google Earth and displays a 3d model of the data.

SUBJECT DESCRIPTORS:

- Specialized information retrieval
- Information visualization
- Geographic visualization

KEYWORDS: Clojure, KML, Visualisation, Aircraft Attitude, XML generation, 3d Modelling

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth (Knuth, 1974)

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Colin Tan for recommending this project to me and for giving me full independence in the implementation. I would also like to thank Mr. Steven Teo of AAIB for his prompt and helpful feedback over the course of the project.

I also would like to extend my thanks to the clojure community in general for releasing such a magnitude of powerful open source libraries that made this project incredibly fun to work on.

Lastly, I would like to thank Mina, my girlfriend, for being patient.

CONTENTS

I THE PROBLEM 1

1 INTRODUCTION 3
- 1.1 Background 3
- 1.2 System Requirements 4

II DESIGNING A SOLUTION 5

2 RENDERING STRATEGIES 7
- 2.1 Keyhole Markup Language 7
- 2.2 Google Earth's Browser Plugin 7
- 2.3 Choosing a rendering strategy 8

3 CHOOSING THE STACK 11
- 3.1 Clojure 11
  - 3.1.1 Functional Programming 11
  - 3.1.2 Libraries 12
  - 3.1.3 Domain Specific Languages 12

4 MODELLING THE PROBLEM 15
- 4.1 Inputs 15
- 4.2 Constraints 15
- 4.3 Flight Path 16
- 4.4 Roll and Heading 16
  - 4.4.1 Distance 17
  - 4.4.2 Bearing 17
  - 4.4.3 Destination Point, given distance and bearing 18
  - 4.4.4 Wings as KML Polygons 18
- 4.5 Thrust 19

III DEVELOPING THE SOLUTION 21

5 IMPLEMENTATION 23
- 5.1 Architecture 23
- 5.2 Parsing 23
- 5.3 Generation 25
  - 5.3.1 math 25
  - 5.3.2 kml 26
- 5.4 Interface 26
  - 5.4.1 UI Flow 27

6 TESTING METHODOLOGY 31
- 6.1 Unit Tests 31
- 6.2 Integration Tests 31
- 6.3 Blackbox Tests 32

7 CONCLUSION 33
- 7.1 Summary 33

7.2 Further Work 33

BIBLIOGRAPHY 35

LIST OF FIGURES

Figure 1 Sample Flight Path, displaying Latitude, Longtitude and Altitude 16
Figure 2 Using wings to visualize roll and heading 17
Figure 3 Using several wings to indicate thrust levels on left and right wings 20
Figure 4 Application architecture as a dependency graph 23
Figure 5 User interface allows users to specify field names in the csv file 25
Figure 6 Users can select which components they want to view 26
Figure 7 An overview of the user interface 29
Figure 8 Success notification 29

LIST OF TABLES

LISTINGS

Listing 1 A simple placemark in raw KML 13

Listing 2 The equivalent placemark with hiccup. Notice the brevity and lack of repetition 13

Listing 3 A sample header from the csv dataset 24

ACRONYMS

DRY Don't Repeat Yourself

API Application Programming Interface

UML Unified Modeling Language

KML Keyhole Markup Language

AAIB Air Accident Investigation Board

Part I

THE PROBLEM

1 INTRODUCTION

1.1 BACKGROUND

Aircraft systems record large quantities of data during flight. A lot of this data is necessary in the system 's proper functioning. It takes an even more significant role when investigating air accidents. As this data establishes the causes behind those accidents, it aids measures to prevent such accidents from occurring. That is the primary reason for why recovery of Flight Data Recorders [1] is usually a very high priority for aircraft investigations (Federal Aviation Administration, 2007).

The data itself features many components tracked over time, including but not limited to the following[2]:

- Latitude
- Longtitude
- Altitude
- Heading
- Pitch
- Roll
- Engine Thrust
- Throttle Lever Position

As is probably evident, analyzing this largely numeric data by itself would be a painstaking task. Primarily because finding patterns in numbers through observation is inherently difficult. Also because the data set is of a substantially large size [3].

Therefore the need for a tool to visualise this data becomes immediately obvious.

The Air Accident Investigation Board (AAIB), a branch of Singapore's Ministry of Transport performs analysis on such flight data, and requested an application that would render relevant flight data

1 Popularly referred to as 'Blackboxes'
2 The following components were part of the sample dataset I was given for testing.
3 I will be cautious before calling it "Big Data".

quickly and in a portable manner.

1.2 SYSTEM REQUIREMENTS

AAIB needed the application to meet certain criterias and contain certain core features. These included:

1. GOOGLE EARTH Compatibility. Google Earth is a popular virtual globe, map and geographic information program (Google Inc., 2013a). It provides detailed 3d projections of Google's satellite images on a spherical globe, overlaid with huge quantities of textual and pictorial data varying from international boundaries to streets. There are many good reasons for this requirement including:
   - Cross Platform. It works and is supported on Windows, OSX and Linux [4]
   - Extensible through the Keyhole Markup Language (KML) [5]
   - Free[6]
   - AAIB's familiarity with it
   - Offline use. This is quite important as internet access is not guranteed on accident sites
2. CSV[7] based input. The system should be able to use large datasets encoded in CSV.
3. The system should display the data in its geographical location as vectors. The visualization should be able to reflect the actual data in an intuitive way. For instance, vectors at each data point should represent roll, heading, engine throttle
4. Users should be able to select which vectors they want to view.
5. The data points should be manually adjustable from within google earth. This is particularly relevant for data points near landing strips, where measurement errors may cause an offset in the rendered visualization

4 Atleast the desktop version is supported on all three major Operating Systems.
5 A lot more on this later
6 As in beer, not speech
7 Comma Separated Values

Part II

DESIGNING A SOLUTION

2 RENDERING STRATEGIES

As with any project, the first few stages of the project involved researching on existing technologies that either implemented the features specified in by AAIB in their requirements, or aided in developing those features. As Google Earth compatibility was a major requirement for the project, my research started with exploring methods to render custom data on Google Earth's platform.

2.1 KEYHOLE MARKUP LANGUAGE

The Keyhole Markup Language (*KML*) is a subset of XML, used to display geographic data in an Earth Browser, like Google Earth. It uses a tag based structure with nested elements that represent certain geometric and descriptive constructs (Google Inc., 2013d). The KML reference is very reasonably documented with sample code demonstrating it's usage.

KML supports rendering the following features:

1. Placemarks
2. Descriptions
3. Ground Overlays
4. Paths
5. Polygons

KML uses latitudes and longtitudes as a coordinate system for positioning elements, which works very well with geographic data. However KML is primarily designed as a means to display data, as opposed to performing interactions with the data. The only interactions it allows are those that Google Earth features, i.e. zooming, panning and moving to a certain point on the globe. Clicking on placemarks with descriptions opens a modal that renders the description, which can be written in a subset of HTML.

2.2 GOOGLE EARTH'S BROWSER PLUGIN

With web browsers becoming increasingly powerful and technologies like WebGL that allow efficient 3d rendering on the browser, Google launched the Google Earth Browser Plugin(Google Inc., 2013c). The

browser plugin by itself allows pretty much the similar core functionality as Google Earth's Desktop Application. The primary difference is that the virtual globe can be embedded within an HTML web page.

By itself the browser plugin may not seem to accomplish much. However Google also released the Google Earth API, a javascript library that provides access to various features in Google Earth. The browser plugin when combined with Google Earth's API becomes a very powerful tool to produce complex applications that render and allow interaction with geographic data.

For instance one of the sample toy applications provided as documentation for the API is a car simulator, where you drive a car over the virtual globe's surface (Google Inc., 2013b).

Such user interactions are possible by combining interactions provided by the DOM, and using them to render models using the Google Earth API.

However, the browser plugin is not a silver bullet. One of it's most lacking features is offline usage. The browser plugin does not work offline, unlike the desktop application which caches data for offline use.

2.3 CHOOSING A RENDERING STRATEGY

After researching on rendering methods available I had to choose which method was a better fit for this project. From my research the following rendering strategies seemed initially viable:

1. Generate KML and use it's Path and Polygon constructs to render the various vectors that AAIB requested
2. Use Google Earth's javascript API to render the same data.

After revisiting AAIB's application requirements, I decided that between producing KML and using Google Earth s Browser Plugin, generating KML was a significantly better option. The rationale guiding the decision was based on the following factors:

- Offline usage. As mentioned before, the browser plugin does not feature offline usage. This is in contrast to rendering KML documents on Google Earth's Desktop client which works well offline. Considering offline usage was an important part of the application requirements, this was a major reason.
- Portability. KML is a well defined format that can be rendered on many other earth browsers. Furthermore, the browser plugin

is only supported for Windows and Mac OS(Google Inc., 2013c). Linux [1] was not yet supported, and I did not want to add that constraint to the application without good reason. In contrast, Google Earth is supported on Windows, Mac OS and Linux.

1 Which I am a huge fan of

3 CHOOSING THE STACK

This decision took a while. And rightfully so, as it would affect all decisions from that point. Choosing the right language and framework could have more than substantial effects on the system's architecture.

After quite a bit of research and debate, I decided to implement the system using CLOJURE, a relatively new lisp class language than runs on the JVM (Hickey, 2008b).

3.1 CLOJURE

The choice of implementing this project in clojure may seem peculiar. However there are a number of features that clojure and it's ecosystem that fit very well with this particular project. Before jumping to those features, a quick overview of clojure is in order.

Clojure is a lisp dialect, created by Rich Hickey. It is a general-purpose, functional programming language that runs on the Java Virtual Machine (JVM) [1]. Clojure focuses on programming with immutable data structures and provides a number of persistent hash-trie based data structures that allow efficient functional programming (Hickey, 2008a).

Clojure was designed to be a real world language, as opposed to a purely academic one. Therefore it integrates seamlessly with it's host environment. For the JVM that means java interoperability is very straightforward (Hickey, 2008c). This also means that all existing libraries and frameworks for Java can be used with Clojure.

The decision to choose Clojure as the main language for implementing this project was based on the following features of clojure

3.1.1 *Functional Programming*

Clojure is foremost a functional programming language. That means the core logic of any application is implemented as a series of pure functions that return immutable values. Pure here means that functions do not return different values for the same set of inputs. The paradigm also involves treating functions as first class members that

1 Clojure runs in other environments like Microsoft's Common Language Runtime (CLR) and Javascript engines as well

can be passed around as values (Michael Fogus, 2011).

As mentioned before, my strategy to rendering the data was to convert it into an appropriate form in KML. This operation in essense is a pure function that consumes a set of inputs and performs several transformations on it to return a KML document.

In other words, there is not state involved. For a given set of data, the output never changes. The transformation operation does not need any side-effects like doing IO or mutating variables.

Furthermore clojure idioms encourage the use of higher order functions that make code succint. I personally enjoy writing programs in a functional style, as opposed to an object oriented style, which despite it's merits can tend to overuse mutation.

3.1.2 *Libraries*

Library support is probably one of the most important factors in deciding the stack for any project. Since part of the system involved generating xml, a fast, xml processing library was a must. Similarly, on the parsing side, even though parsing csv files is trivial, the parsing needed to be done in a performant manner, especially when the data sets might be very large. Also, the input data format could change in other scenarios, and having a decent set of parsing tools for more advanced grammars (JSON, XML) makes a lot of sense.

On both counts clojure scores very well. In terms of xml processing, clojure.data.xml provides a decently documented, clean api which can lazily parse and generate large xml documents. Under the hood its uses Java's STAX library (Clojure Org., 2013a).

But in addition to xml generation and parsing, clojure also provides a very neat library called clojure.data.zip for performing queries and transforms on generic tree like structures. clojure.data.zip, as the name suggests uses Zippers to functionally traverse and modify trees (Clojure Org., 2013b).

3.1.3 *Domain Specific Languages*

Building Domain Specific Languages (DSLs) is where lisp like languages are reputed to shine. Lisp's homoiconicity, and the ability to treat code as data helps quite a bit here. Clojure is no exception to this rule, and features many rich DSLs.

The need for a DSL particularly arose when dealing with XML generation. As the application required generating KML, which is a subset of XML, finding a tool that eases it's generation was paramount. XML by itself is not "designed for humans" (Atwood, 2008). The syntax is verbose, repetitive and prone to errors.

Representing XML data as raw strings can be incredibly messy and error prone, as there is no way to gurantee the validity of document structure. Forgetting or misplacing closing tags can lead to bugs, that would be very difficult to find if the generated XML document is complex. Considering the application was meant to generate reasonably complex KML documents, using raw strings was to be avoided at all costs.

Enter Hiccup, a very popular DSL that represents XML using clojure's vectors and hash-maps (Reeves, 2013). Although originally just designed for HTML, the DSL became so popular that most of clojure's XML libraries support it.

The following snippet compares raw KML with the equivalent Hiccup representation:

Listing 1: A simple placemark in raw KML

```
<Placemark>
  <name> Simple Placemark  </name>
  <Polygon>
    <extrude>0
    </extrude>
    <outerBoundaryIs>
    </outerBoundaryIs>
  </Polygon>
</Placemark>
```

Listing 2: The equivalent placemark with hiccup. Notice the brevity and lack of repetition

```
[:Placemark
  [:name ''Simple Placemark'']
  [:Polygon
    [:extrude ''0'']
    [:outerBoundaryIs]]]]
```

As is evident, the sample using hiccup is far more succint. This becomes even more evident when the size of the document increases.

However hiccup is not only useful because of it's brevity. It's killer feature is the fact that the representation is simply a clojure vector,

which can be manipulated as a proper tree like data structure. This reduces XML generation to manipulating clojure vectors, which is signifigantly easier.

Another major feature that hiccup brings is composibility. Because I just have to deal with clojure vectors, I can make small functions that produce base components of the document and compose them together to form more advanced forms. This is very powerful and much more elegant than the naive approach of concatenating XML strings together.

4

MODELLING THE PROBLEM

As mentioned before, the core function of the application was to convert periodic flight data into a renderable 3d model. This section will detail the approach taken to model the various aspects of the flight data.

4.1 INPUTS

The dataset AAIB provided, contained the following measurements, recorded over time t.

1. Magnetic Heading $\theta_m(t)$. Measures the direction of the aircraft with respect to the north pole. Measured in degrees
2. Pressure Altitude $a(t)$. Indicated altitude, measured in feet
3. GPS Latitude $\phi(t)$. Measured in degrees
4. GPS Longtitude $\lambda(t)$. Also measured in degrees
5. Roll Angle $\theta_r(t)$. Also measured in degrees
6. Engine Thrust $e_i(t), 0 < i \leqslant 4$. Measured as a percentage

The dataset had other fields as well, but they were not used for the model implemented

4.2 CONSTRAINTS

A constraint that choosing KML placed on the model was that any coordinate could only be represented withe following dimensions(Google Inc., 2013d):

1. Latitude (ϕ)
2. Longtitude (λ)
3. Altitude a

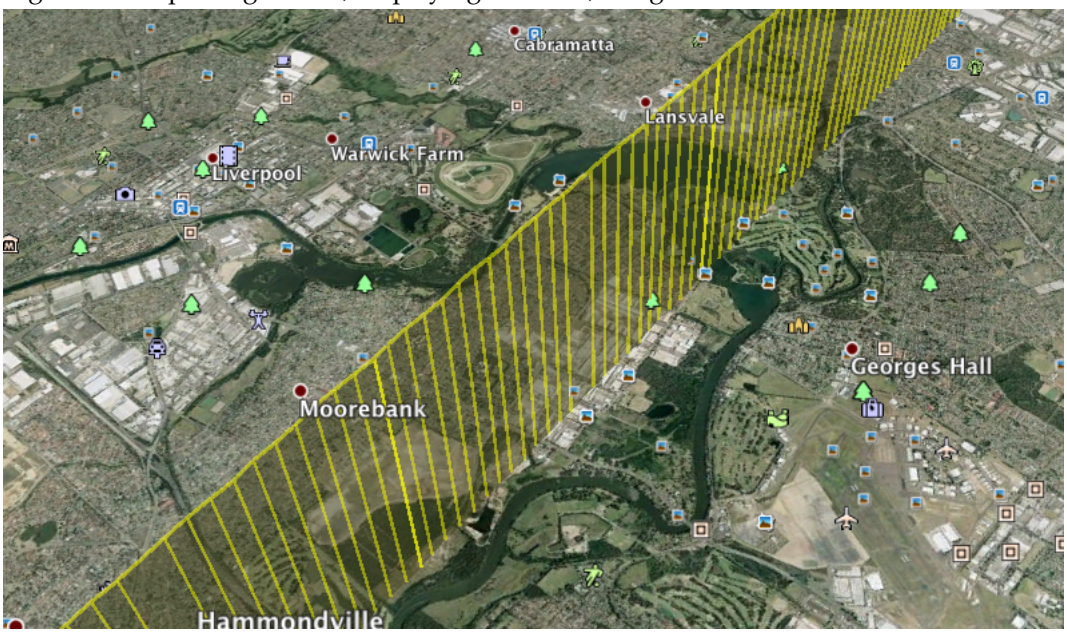
This implied that plotting polygons with KML would require converting to and fro between cartesian and spherical coordinate systems.

4.3 FLIGHT PATH

Plotting the flight path was the most straightforward operation. The flight path was to be modelled using KML's Line type, which just required sets of Latitude, Longtitude and Altitude readings

$$FlightPath = \{\{a(t), \phi(t), \lambda(t)\} | t_{start} < t \leqslant t_{end}\}$$

Figure 1: Sample Flight Path, displaying Latitude, Longtitude and Altitude



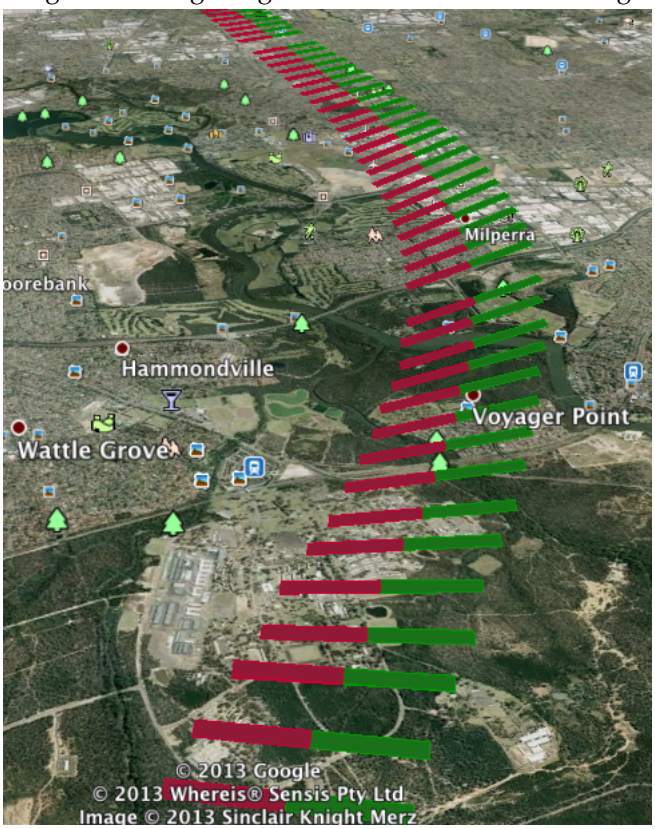
4.4 ROLL AND HEADING

Representing the roll and heading with KML was more complex. The central idea was to model the aircraft's "wings", which would give a visual representative of roll by assessing the roll angle of the wing. Modeling the wings would also give an indication of the heading, if the wings are assumed to be straight lines. The heading would always be perpendicular to the wings.

The following figure should make this idea clearer:

The wings could be modelled using KML's Polygon type. However, as mentioned before, we can only use latitude,longitude and altitude as coordinates for KML's types. Therefore to make the modelling easier, the following concepts are necessary:

Figure 2: Using wings to visualize roll and heading



4.4.1 *Distance*

Finding the distance between two latlong points is an essential operation. For this application I used the Haversine Formula, which works was follows(Veness, 2013):

$$a = \sin^2(\Delta\phi/2) + \cos(\phi_1)\cos(\phi_2)\sin^2(\Delta\lambda/2) \tag{1}$$

$$d_H = 2R_{earth}\arctan(\frac{\sqrt{a}}{\sqrt{1-a}}) \tag{2}$$

Here d_H refers to the Haversine distance between two latlong points $\{\phi_1, \lambda_1\}$ and $\{\phi_2, \lambda_2\}$ and R_{earth} refers to the Earth's radius.

The Haversine distance gives the shortest distance between both points over the Earth's surface. It can be thought as being equivalent to the "as the crow flies" distance between two points.

4.4.2 *Bearing*

Bearing θ between two latlong points $\{\phi_1, \lambda_1\}$ and $\{\phi_2, \lambda_2\}$ can be calculated with the following method(Veness, 2013):

$\theta = \arctan(\frac{\sin(\Delta\lambda)\cos(\phi_2)}{\cos(\phi_1)\sin(\phi_2) - \sin(\phi_1)\cos(\phi_2)\cos(\Delta\lambda)})$
Bear in mind, this bearing is the initial bearing, also referred to as the forward azimuth, which if followed in a straight line would take one from the first point to the second. By straight line I mean a stright line over the Earth's surface, also referred to as a great circle.

4.4.3 *Destination Point, given distance and bearing*

Probably the most useful concept in terms of generating KML. Representing any polygon in KML would require calculating a point's latlong coordinates when given a distange range and bearing.

To calculate the coordinates $\{\phi_2, \lambda_2\}$ located a distance d from point $\{\phi_1, \lambda_1\}$ at a bearing of θ, the following can be used(Veness, 2013):

$$\phi_2 = \sin(\phi_1)\cos(d/R_{earth}) + \cos(\phi_1)\sin(d/R_{earth})\cos(\theta) \quad (3)$$

$$\lambda_2 = \lambda_1 + \arctan(\frac{\sin(\theta)\sin(d/R_{earth})\cos(\phi_1)}{\cos(d/R) - \sin(\phi_1)sin(\phi_2)}) \quad (4)$$

One thing to note though is that these equations assume altitude from the earth's surface to be negligible when compared to the Earth's radius. Since this particular application mostly deals with commercial aircraft which do not cruse above 18 km, which is significantly lesser than the Earth's Radius (6,371 km)

4.4.4 *Wings as KML Polygons*

With equations 3 and 4 above, modelling wings via KML becomes easier. From a given point, it should be easy to obtain the coordinates of a rectangle of a particular width and height.

To illustrate this further, lets define the following function

$$destinationPoint(point, \theta, d, a) \rightarrow \{\phi, \lambda, a\} \quad (5)$$

It takes a latlong coordinate $point$, an initial bearing θ, a distance d and altitude a and returns a destination coordinate using equations 3 and 4.

To define a wing, lets assume we have a starting coordinate $\{\phi, \lambda, a\}$, a width of $2w$, a height of $2h$, a heading $\theta_{heading}$ and a roll angle θ_{roll}

$$\beta_1 = \arctan(w/h) \quad (6)$$

$$\beta_2 = 2(\pi/2 - \beta_1) \tag{7}$$

$$r = \sqrt{w^2 + h^2} \tag{8}$$

$$\delta a = \sqrt{2} r \sin(\theta_{roll}) \tag{9}$$

δa measures the change in altitude required to represent the roll

Finally, lets define an array $Wing$ which consists 4 vertices to plot the rectangle

$$Wing[0] = destinationPoint(\{\phi, \lambda\}, \beta_1 + \theta_{heading}, r, a + \delta a) \tag{10}$$

$$Wing[1] = destinationPoint(\{\phi, \lambda\}, \beta_1 + \beta_2 + \theta_{heading}, r, a + \delta a) \tag{11}$$

$$Wing[2] = destinationPoint(\{\phi, \lambda\}, \beta_1 + \pi + \theta_{heading}, r, a - \delta a) \tag{12}$$

$$Wing[3] = destinationPoint(\{\phi, \lambda\}, \beta_1 + \beta_2 + \pi + \theta_{heading}, r, a - \delta a) \tag{13}$$

After this, $Wing$ can easily be rendered as coordinates of a KML Polygon

4.5 THRUST

Visualising thrust is relatively much easier. Since we already have a method for rendering wings of a certain width and height, thrust can be visualized by simply adjusting the width and height of the wing and making it proportional to thrust of engines on that wing.

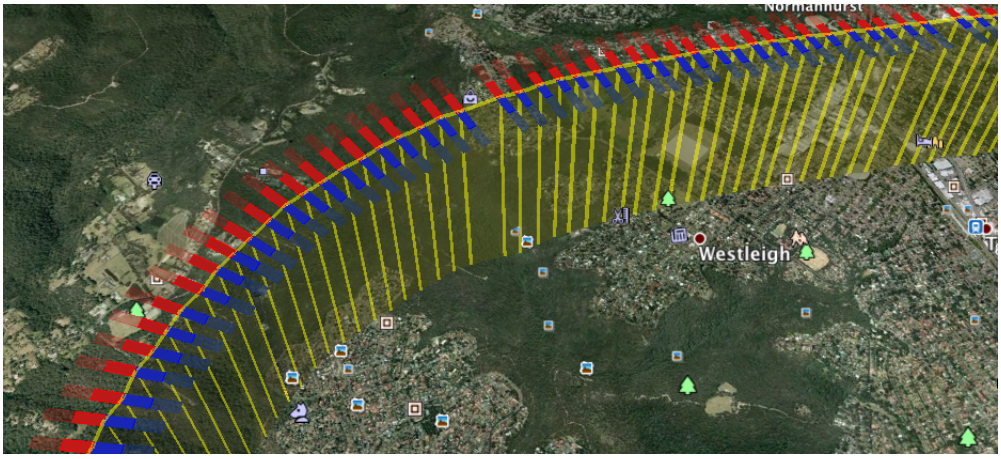
The only change required would be to represent left and right wings as two separate rectangles. The reason for this is because many commercial aircraft have engines on both wings. The dataset provided by AAIB had 4 engines, two on each wing, which may have

differing thrust values.

The figure below should make this idea clearer. The rectangles with the darker color represent thrust on both wings. They are positioned over rectangles of a lighter color, whose size would correspond to 100% thrust.

This allows the user to get a better idea of what the thust level at that point in the flight was.

Figure 3: Using several wings to indicate thrust levels on left and right wings



Part III

DEVELOPING THE SOLUTION

5 IMPLEMENTATION

5.1 ARCHITECTURE

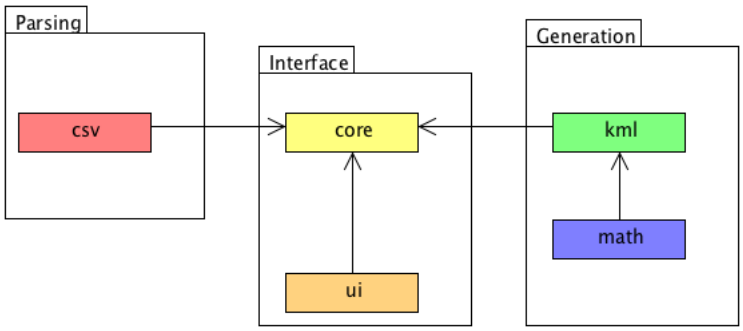


Figure 4: Application architecture as a dependency graph

The application was divided into 5 main modules, or namespaces in clojure speak, with each namespace containing functions that perform a particular task.

As evident by the figure above, these namespaces were grouped into three broad areas; parsing, interfacing and generation. The primary motivation for having all this classification was to maintain decoupled code. The modularity also helped with keeping the code easier to extend.

For example, if the input format was to change, the only module effected would be the csv module. The kml, math and ui namespaces would be unaffected by this change.

Another broad aspect of the architecture was to keep the interfaces between the modules well defined. Therefore even though the implementation might change, things would not break.

5.2 PARSING

As mentioned in the requirements, the application should be able to read input from a csv file. Parsing the data by itself was pretty straightforward. Clojure comes with a very performant csv parser called clojure.data.csv which returns a sequence of vectors, represent-

ing the columns and rows in the csv data set.

The parser does need logic to ignore comments, as clojure.data.csv does not recognize them. Nor does it ignore whitespace. Therefore I needed to add some simple logic to filter out whitespace and comments.

A very important part of the dataset is the data header. This usually is the first line of the file, if whitespace and comments are ignored. The header specifies which column represents what data.

Therefore the csv namespace offers a function called `read` which takes a string or a buffered reader, performs the parsing and returns a vector of hash-maps. Each hash-map represents a row, and each value in hash-map represents the value at that row for a particular column. The column names are kept as keys during this representation.

As the mapping between column names in the data set and what they represent in the application is not fixed,it would not be wise to hardcode that mapping in the application. The CSV file has no gurantees that the headers would remain the same for all datasets.

Listing 3: A sample header from the csv dataset

```
"Time","Magnetic Heading (DEGS)","Pressure Altitude (feet)","GPS
    Latitude (degs)","GPS Longitude (degs)","AIR GROUND (0-AIR,1-
    GND)","Throttle Lever Position Engine 1 (DEGS)","Throttle
    Lever Position Engine 2 (DEGS)","Throttle Lever Position
    Engine 3 (DEGS)","Throttle Lever Position Engine 4 (DEGS)","
    Roll Angle (DEGS)","Pitch Angle (DEGS)","N1 Actual Engine 1
    (\%)","N1 Actual Engine 2 (\%)","N1 Actual Engine 3 (\%)","N1
     Actual Engine 4 (\%)","UTC Time  (hh:mm:ss)"
```

The application tackles this problem in two ways:

1. The user interface requests the user to fill in the mapping (see figure below).

2. The parser guesses which headers strings correspond to what data. The guessing logic for now is very simple. It simply checks if a field in the header contains a certain substring which would recognize that field. This is not a fool proof method by any means, but helps in reducing work for the user by auto filling the mapping section in the user interface.

To perform the guessing, the csv namespace exposes a function called `analyze-headers!`

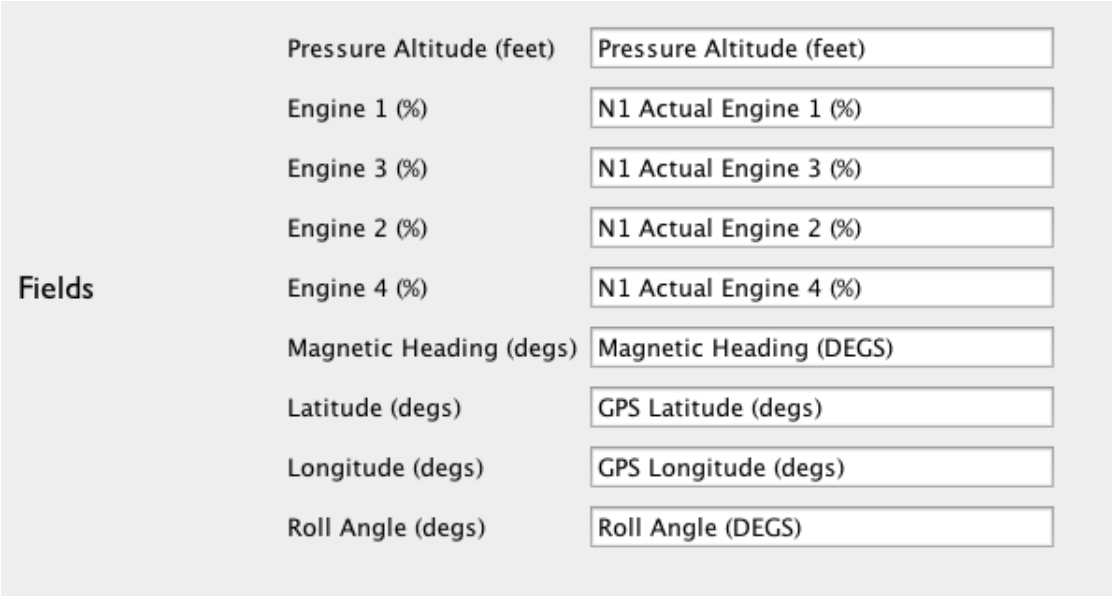


Figure 5: User interface allows users to specify field names in the csv file

As clojure sequences are lazy data structures, I do not need to worry about heap overflows, if the data size increases. That is the fundamental reason why clojure.data.csv returns a sequence of rows, intead of a vector, which is not lazy.

5.3 GENERATION

There are two main namespaces that classify within generation.

5.3.1 *math*

As should be evident by the name, this namespace largely implements the mathematical models mentioned in the previous chapter. The main functions it exposes include `right-rect-from-edge` and `left-rect-from-edge`.

`left-rect-from-edge` takes a point, width, height, heading and roll angle to produce a rectangle that represents the left wing of an aircraft[1]. Similarly `right-rect-from-edge` does pretty much the same thing for the right wing of the aircraft.

1 When viewed from behind

5.3.2 *kml*

This namespace contains functions that do the bulk of the KML generation. The main interface here is the `render` function which generates the KML document and emits the result to a file.

The hiccup DSL is used very heavily here, along with clojure.data.xml for speedy emission. Hiccup is primarily used to build base XML components which are fed coordinates from calling functions from the `math` namespace.

KML allows specifying styles to elements, which works like a bit like CSS. KML elements can have ids, which Style objects can refer to when they specify styles like line and fill color. Because user's would want the ability to customize colors, the `render` function also takes a style configuration as an argument.

AAIB also specified in the user requirements, that users should be able to select and unselect various elemnts in the visualisation. This taken care of within this namespace through using KML's Folders to structure the document in a user friendly way.

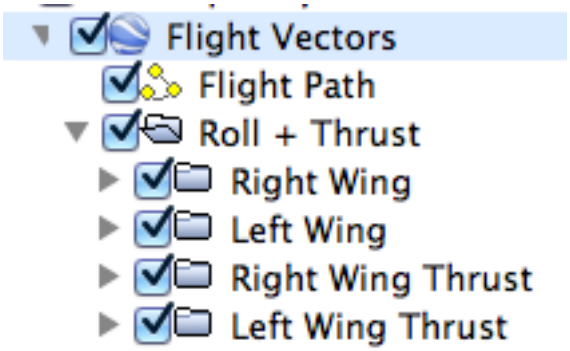


Figure 6: Users can select which components they want to view

5.4 INTERFACE

The application needs to be able to effectively interact with users. Since the core functionality of the application is very specific, the set of possible user interactions is actually not very large. The application features the following interactions:

- specify input file path
- specify output file path

- fill field names from the CSV dataset, if they are guessed incorrectly
- assign colors to the each of components
- add engine configuration

Using clojure meant that Java's swing libraries were available for writing UI code. However swing by itself is incredibly verbose. Although clojure supports interoperability with Java, writing clojure code in a Java-esque fashion would defeat the purpose of using clojure in the first place.

Enter seesaw, a well documented clojure library and DSL that allows developers to construct swing based applications without having to deal with swing directly (Ray, 2012).

Seesaw's allows the creation of swing objects in a very succint fashion. Because of clojure's nature of being a lisp and running at a REPL, seesaw allows a very fast feedback loop when designing a user interface, as seesaw functions can be invoked at the REPL.

In addition to just creating swing object, seesaw also allows querying the tree of UI elements for specific elements. This API is inspired from CSS selectors on web browsers. One can query elements by their ids or their classes.

I made heavy use of seesaw when writing the code for the user interface. Which is also why I was able to keep the `ui` namespace under 200 lines of code.

5.4.1 *UI Flow*

1. The user launches the application
2. He/she selects an input csv file. This can be done by either writing the path to the file manually in the text box, or clicking on a button that launches a file picker. As soon as the input file is selected, the application analyses the file's headers and auto fills the fields section.
   Under the hood, this launches a call to the `csv/analyze-headers!` on a separate thread. This is so any processing does not freeze the interface.
3. The user enters the path he/she wants to output to.
4. The user fills up any missing fields, or corrects fields that may have been detected wrong.

5. The user chooses colors for each of the components. Clicking on colored button next to the component name launches a color picker, where the user can choose a new color for that component.

6. Lastly, the user fills up the engine configuration. This specifies which engine is on which wing.

7. The user hits submit. The KML generation is triggered asynchronously, once again to prevent the UI from freezing.

8. If the file, the user selected was valid and contained the fields specified, the user should see a label with "Success!". Otherwise it pops an alert with an error message.

Figure 7: An overview of the user interface

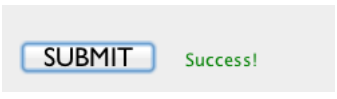


Figure 8: Success notification

6 TESTING METHODOLOGY

Testing for the application took place in 3 main stages. Clojure comes built in with a test framework called clojure.test which was mainly used.

6.1 UNIT TESTS

Unit tests were particularly important for functions within the `math` namespace. The primary reason was because the entire generation component depended on mathematical model being implemented properly.

The unit tests also gave an indication of how numerically stable the computations were, especially when dealing with very small distances, or angles that were multiples of π. As clojure automatically shifted to using java's BigDecimal type when dealing with high precision numbers, thankfully the calculations did not face major numerical instabilities.

Unit tests were also used for the parsing stage. This stage was even easier to unit test, primarily because the functionality was relatively simple. The major edge cases I had to consider were cases where input was malformed or corrupted, and to fail gracefully when encountering such cases.

However, unit testing the KML generation was not straight forward. The primary reason is that because the application generates a visualization, it is difficult to assess it's validity programmatically, aside from checking for trivial things like coordinates being non empty.

6.2 INTEGRATION TESTS

Integration tests fell under two main categories:

1. Testing the parsing and generation components together. Inputs were fed into both components without the UI to test whether these two subsystems integrated properly. Part of this test included checking for correct error handling when given incorrect or corrupt csv inputs. It also included checking for visualizations when several data points were missing.
   Sadly however, checking the KML output could not be automated and had to be done manually for this stage.

2. Testing all the UI components with the UI flow without coupling them to the application core. Seesaw's selectors made this much easier than I anticipated.

6.3 BLACKBOX TESTS

Testing the entire system was once again something that could not be automated, except for trivial cases, as the system produced a visualization that necessitated manual inspection.

To make black box testing easier, I made a list of test cases that I would manually replicate, where I would test for incorrect input files, mistyped field names, and impossible engine configurations, and check whether the UI displayed errors in these cases.

One limitation to extensive testing was that I was only given one dataset by AAIB to use for development. Having more datasets would have allowed me to test for more cases.

7 CONCLUSION

7.1 SUMMARY

The project started off with what seemed to be a complex problem. The problem was to map raw aircraft attitude data into a form that could be visualised easily. What really helped here though was having a good mental image of what I was expected to render.

The next stage was to model this mental image when given the constraints of the output form. As I had chosen KML to render this data, I had to represent whatever images I had in mind, in terms of KML elements. This was intially difficult as I had to visualize simple geometric constructs like lines and polygons in spherical coordinates.

However, after researching and finding ways to transform spherical coordinates by distance and bearing, the next steps were easy.

Implementation wise, building this system in a language that favored REPL based development, helped me experiment and move quickly with ideas. Furthermore, using well developed DSLs like Hiccup in Clojure made my work significantly easier.

7.2 FURTHER WORK

Although the project fulfilled the features AAIB requested, it can be improved further. A few features, in my opinion that the current visualisation lacks are:

- Displaying an aircraft's pitch. This is actualy quite simple to add. For each data point, one could add a line representing the pitch angle, that would run through the aircraft's lateral axis.
- Displaying an aircraft's yaw. This would be slightly difficult to visualize with the current scheme of things. The difficult would lie in visually differentiating between the yaw and heading, which currenly display on the same axis.
- Finding a method to overlay data when the user places his mouse over a data point. I actually tried implementing this, but because KML is not designed for complex interactions like mouseover events, this particular feature is not possible with just KML. An alternative could be to launch a description modal

when clicking a data point, but with a large number of data points this could look very messy.

BIBLIOGRAPHY

Atwood, J. (2008). Xml: The angle bracket tax. http://www.codinghorror.com/blog/2008/05/xml-the-angle-bracket-tax.html.

Clojure Org. (2013a). Github: clojure.data.xml. https://github.com/clojure/data.xml.

Clojure Org. (2013b). Github: clojure.data.zip. https://github.com/clojure/data.zip.

Federal Aviation Administration, F. (2007). Flight data recorder systems.

Google Inc. (2013a). Google earth. http://www.google.com/earth/explore/products.

Google Inc. (2013b). Google Earth API Developer's Guide. https://developers.google.com/earth/documentation/.

Google Inc. (2013c). Google Earth Plug-in. http://www.google.com/earth/explore/products/plugin.html.

Google Inc. (2013d). KML Reference. https://developers.google.com/kml/documentation/kmlreference.

Hickey, R. (2008a). Clojure, Data Structures. http://clojure.org/data_structures.

Hickey, R. (2008b). Clojure, features. http://clojure.org/features.

Hickey, R. (2008c). Clojure, java interop. http://clojure.org/java_interop.

Knuth, D. E. (1974). Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673.

Michael Fogus, C. H. (2011). *Joy of Clojure*. Manning, 1st edition.

Ray, D. (2012). Seesaw turns the horror of swing into a friendly, well-documented, clojure library. https://github.com/daveray/seesaw.

Reeves, J. (2013). Hiccup. https://github.com/weavejester/hiccup.

Veness, C. (2013). Calculate distance, bearing and more between latitude/longitude points. http://www.movable-type.co.uk/scripts/latlong.html.