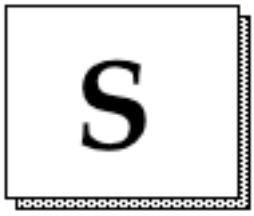


FOURTH YEAR ELECTRICAL AND COMPUTER SYSTEMS ENGINEERING
4th YEAR THESIS PROJECT ECE

ECE4912 – Thesis Project

UAV Ridge Soaring

Ashley Goodwin

DEPARTMENT OF ELECTRICAL AND COMPUTER SYSTEMS ENGINEERING
MONASH UNIVERSITY, CLAYTON, VICTORIA 3168, AUSTRALIA

MONASH UNIVERSITY
FACULTY OF ENGINEERING
ASSESSMENT COVER SHEET

Surname(s)	Goodwin		
Given names	Ashley		

Student's I.D. number	18574661

Unit code and name	ECE4912 Thesis Project 2

Title of assignment	UAV Ridge Soaring

Name of lecturer	Prof. G. Egan

Class day/time	

Due Date	Date submitted

All work must be submitted by the due date. If an extension of time is granted this must be specified with the signature of the lecturer, tutor or demonstrator concerned.

Extension granted until (date) …… Signature of lecturer/tutor
……..……

Please note that it is your responsibility to retain copies of your assessments.

Plagiarism and Collusion are methods of cheating for the purposes of Monash Statute 4.2 – Discipline.

Plagiarism: Plagiarism means to take and use another person's ideas or work and pass these off as one's own failing to give appropriate acknowledgement.
This includes material from any source – published and unpublished works, staff or students, the Internet.

Collusion: Collusion is the presentation of work which is the result in whole or in part of unauthorised collaboration with another person or persons.

Where there are reasonable grounds for believing that plagiarism has occurred, this will be reported to the Chief Examiner, who will disallow the work concerned by prohibiting assessment and inform the student and Associate Dean (Teaching).

Student's statement:
I/We certify that I/we have not plagiarised the work of others or participated in unauthorised collusion when preparing this assignment.

Signature(s) …… ……

…… ……

Executive Summary

Acknowledgements

I would like to show my appreciation firstly to Professor Greg Egan for the supervision and support that made this thesis possible. I would also like to thank Lawrence Goodwin for his technical advice and Jane Waugh for her moral support.

Table of Contents

Table of Figures

1. Project Description

An Unmanned Air Vehicle (UAV) keeps itself in the air by using petrol or electric motors to propel air over their wings, creating lift. The aim of this project was to enable a UAV to increase the amount of time it spends aloft by taking advantage of any lift or rising air that occurs naturally due to the terrain – in particular, ridge lift, which is formed when wind is deflected upwards by hills, mountains, and etcetera. By using the rising air, a UAV is less reliant on its own motor, extending the life of the batteries and increasing the possible flight time. For the project, development of a software guidance system was required, which with no prior knowledge of the layout of the terrain gives advice to the (auto) pilot as to which direction of travel is more ideal.

The guidance system receives information from the UAV's onboard instrumentation – GPS, altimeter, etc. – and overtime builds a map of the airflow conditions. The information is stored using location as the index. The purpose of remembering this information is to allow the guidance system to evaluate the possible amount of lift in one area versus another.

Also required in the project is the writing of a conference paper.

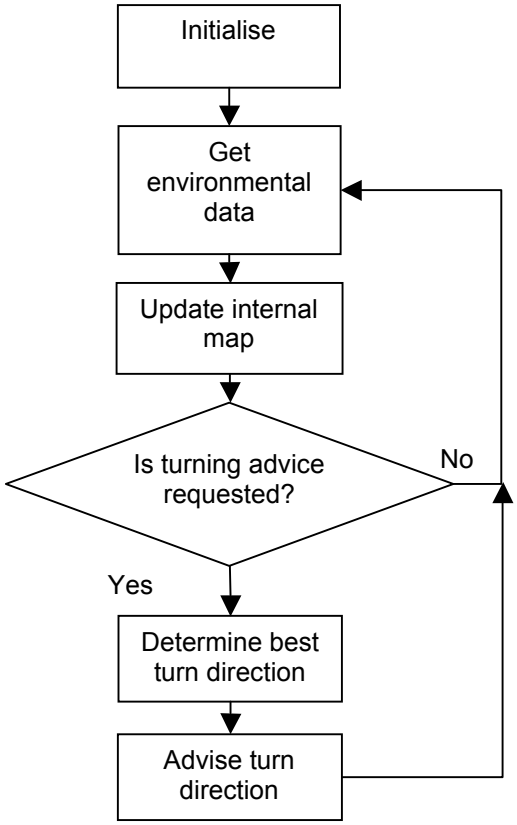


Figure 0. Flow diagram of Turn Direction Adviser.

2. Software Design

The software written to accomplish the project consists of two components: the Simulation System and the Advice System. The Simulation System exists to replicate the real-world environment a UAV would experience. The Advice System uses information supplied by the Simulation System to complete the project objectives; namely provide advice on ideal travel directions.

The Simulation System provides a graphical display in order to show the UAV's flight. OpenGL was chosen to facilitate these graphics.

The software is deliberately designed to be highly structured. Functions that belong to the same logical group are contained in the same file to provide easy navigation. This method allows easy debugging and exposes the logical structure of the software.

2.1. *Advice System Design*

The Turn Advice system operates according to the flowchart shown in Figure 0.

In order to give advice, the system uses present and prior data obtained from the UAV instrumentation. The data is stored in a map that is indexed by position. Rather than store the map as a contiguous

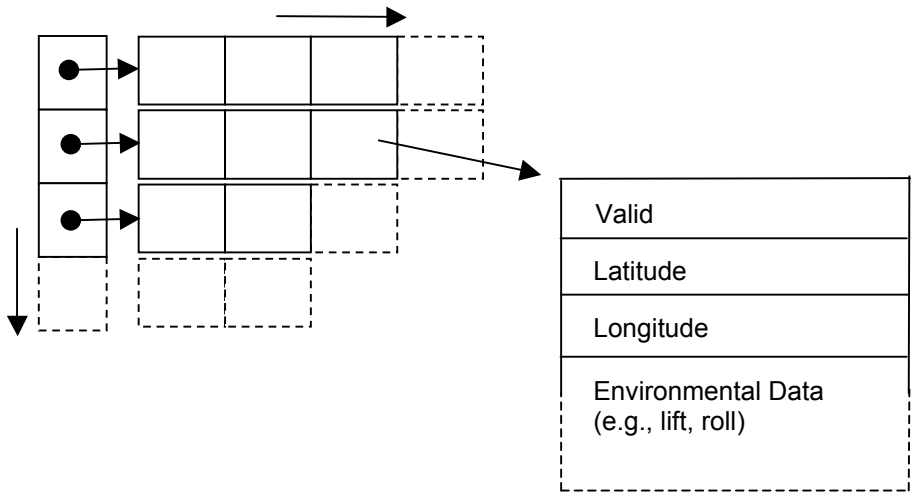


Figure 0. Diagram of internal map and its possible elements.

array, the rows are divided such that each row is an individual array. The map maintains pointers to these rows. The design choice was made in order to minimise the cost of moving the map. For example, if the locations the map covers are to be shifted vertically by one row, it is a simple operation to discard the out-of-range row, shift the pointers to the remaining rows and create a new row. The order of this procedure is *O(n)*. Had a contiguous array been used, it would be necessary to move every element of every row to its new position explicitly. The order of doing this is $O(n^2)$, a much more severe penalty. It is noted that horizontal moves still attract a cost of $O(n^2)$. Using a linked-list could reduce this; however this improvement must be balanced against the added complexity of element traversal. It was decided that there would be no benefit in using a linked-list map.

The information each element of the map holds is stored in a structure. The actual makeup of the structure depends upon the needs of the Advice Algorithm. No matter what the needs of the algorithm, the structure will contain a Boolean field indicating whether the information contained is valid.

A diagram illustrating the form of the map and a possible implementation of the elements' structure is show in Figure 0.

2.1.1. Simple Turn Adviser

The simplest method to determine on which side of the UAV has the greatest instantaneous lift is too use the roll of the aircraft. Across the wingspan of the UAV, lift is non-uniform; that is, the amount of lift pushing up one wing will be more than the amount pushing up the other. The effect of one wing being pushed harder than the other is to introduce a roll into the UAV (as seen in Figure 0). By inspecting the roll, the Advice System can determine which wing was subject to

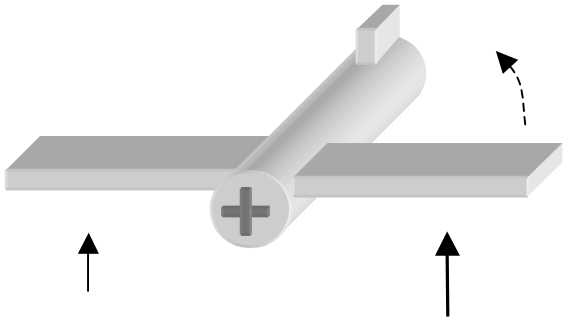


Figure 0. The effect of differing levels of vertical air on the UAV. The air on the right is rising faster and causes the UAV to roll anticlockwise.

greater lift. By advising the UAV to turn towards the side with greater lift, the UAV associates itself with the local lift maxima.

Significantly, the Simple Turn Adviser doesn't not remember or reuse any information obtained from the UAV's instrumentation; it does not *learn*.

2.1.2. Advanced Turn Adviser

Unlike the Simple Adviser, the Advanced Turn Advice system relies solely upon remembering environmental conditions at specific locations. The system uses what it has learnt to direct the UAV. The basic action of the algorithm implemented is to search the map starting from the UAV's current position and moving radially outward. Figure 0 shows one quadrant of the search paths used – the other quadrants can be obtained by reflecting the figure in each of the axes.

Based upon the cruise speed of the UAV, and the radius of its turning circle at that speed, the algorithm calculates the area of sky adjacent to the UAV that cannot be reached (through inability to turn tightly enough). Because this region of sky cannot be reached, the algorithm does not test it.

The algorithm divides the space around it into 30° sections and accumulates the lift from each search path into its accompanying section. Once all the paths have been processed, the centre angle of the section with the greatest accumulated lift is decided to be the most optimal angle of travel. Should all the sections have an aggregate lift of zero, then the algorithm will determine the section with the most negative lift and advise that the UAV should turn *away* from that section. If there are no sections whatsoever with any kind of lift, positive or negative, the algorithm advises the UAV to continue upon its current heading.

The Advanced Turn Adviser contains a "safety" option: should the UAV stray too far from the centre of the map, the turn adviser will instruct the UAV to head back to the centre. The distance the UAV can travel from the centre was dubbed the Escape Distance, and is specified in the

configuration file. It is usually set to a value large enough to allow the UAV to roam freely over the entire map but can be set to zero to disable it.

The number of paths, or angles, tested in each quadrant is specifiable in the 'Advice.h' header file (see Appendix 8.1), as is the distance along the search path to test.

2.1.3. Preadvice Pattern Flight

When the Advanced Advice System is called upon to operate, it can immediately attempt to give advice or can firstly fly a predefined pattern – the Search Pattern.

The purpose of instructing the UAV to fly the search pattern is to partially fill the Advice System's internal map of lift locations. Without this information, the system cannot give Advanced advice.

The shape the UAV follows when flying the search pattern is a clockwise decreasing spiral. The shape can be seen in Figure 1. The search pattern starts by following the outside of the area covered by the internal map, and proceeds to spiral inwards with user-definable spacings between successive spirals. The spacing can be modified by changing the simulation configuration file.

2.2. *Simulation System Design*

The Simulation System provides a 3D world within which to test the Advice Systems and its algorithms. It contains a 3-dimensional video display to allow the UAV to be observed. The terrain of the simulated world is read in from a text file and generated dynamically. This allows the terrain to be edited simply and quickly, without the need to use a special editor. Many of the properties of the simulation, such as wind strength and direction, are specified in the same text file as the terrain. There exist a number of free, open-source flight simulators that can be downloaded from the internet
downloaded from the internet. Two such packages that were considered for use in this project were:

- Flight Gear[1]; and,
- Slope Soaring Simulator (SSS)[2].

Flight Gear is a well recognised flight simulator which many people may be familiar with. SSS is less well known, and is specifically designed for UAV and RC aircraft. I investigated the possibility of using Flight Gear as the simulation base, however had to discard it as once it was compiled (an effort in itself) it took approximately 10 minutes to load its data files and begin simulating. The lack of an accelerated graphics card in my computer also meant a very low frame rate. The difficulties with SSS began much earlier, with an inability to compile the package despite my best efforts. The SSS source code for reading a configuration file was used, however. Also used was code from [3] for determining the shortest distance from a point to a line.

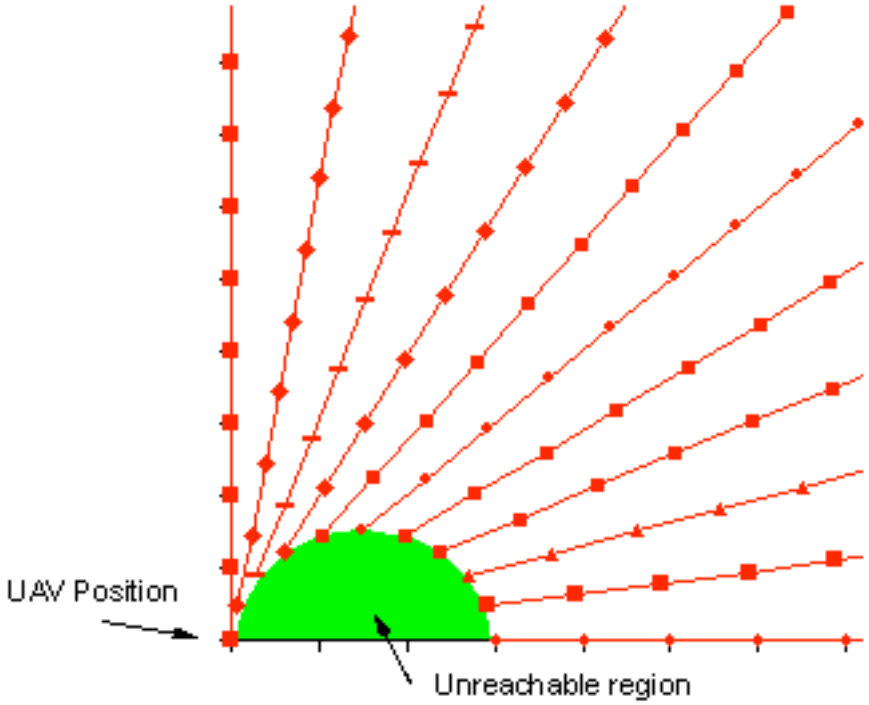


Figure 0. One quadrant of the search pattern, showing the test points and the unreachable region.

In the development of the project, three types of lift model have been used. They are:

- Thermal;
- Simple ridge; and,
- Advanced ridge.

Only the advanced ridge is implemented in the final version of the simulator.

The thermal and simple ridge models were constructed using sinusoidal cross-sections, as diagrammed in Figure 1. Whilst these models were useful in developing the Advice flowchart and the Simple adviser, they were of limited use due to their 2D nature. In order to operate fully in 3D, a more advanced ridge model was needed.

2.2.1. Terrain & the Advanced Ridge Model

Highly accurate three-dimensional models of the behaviour of air as it flows over a ridge are complex and involve many parameters. [4] tells as that the direction of airflow at a location depends on:

- Wind velocity
- Static stability of the air
- Terrain shape

Time is also involved; as the first two of the above parameters are not constant (it could be argued the third is also not constant).

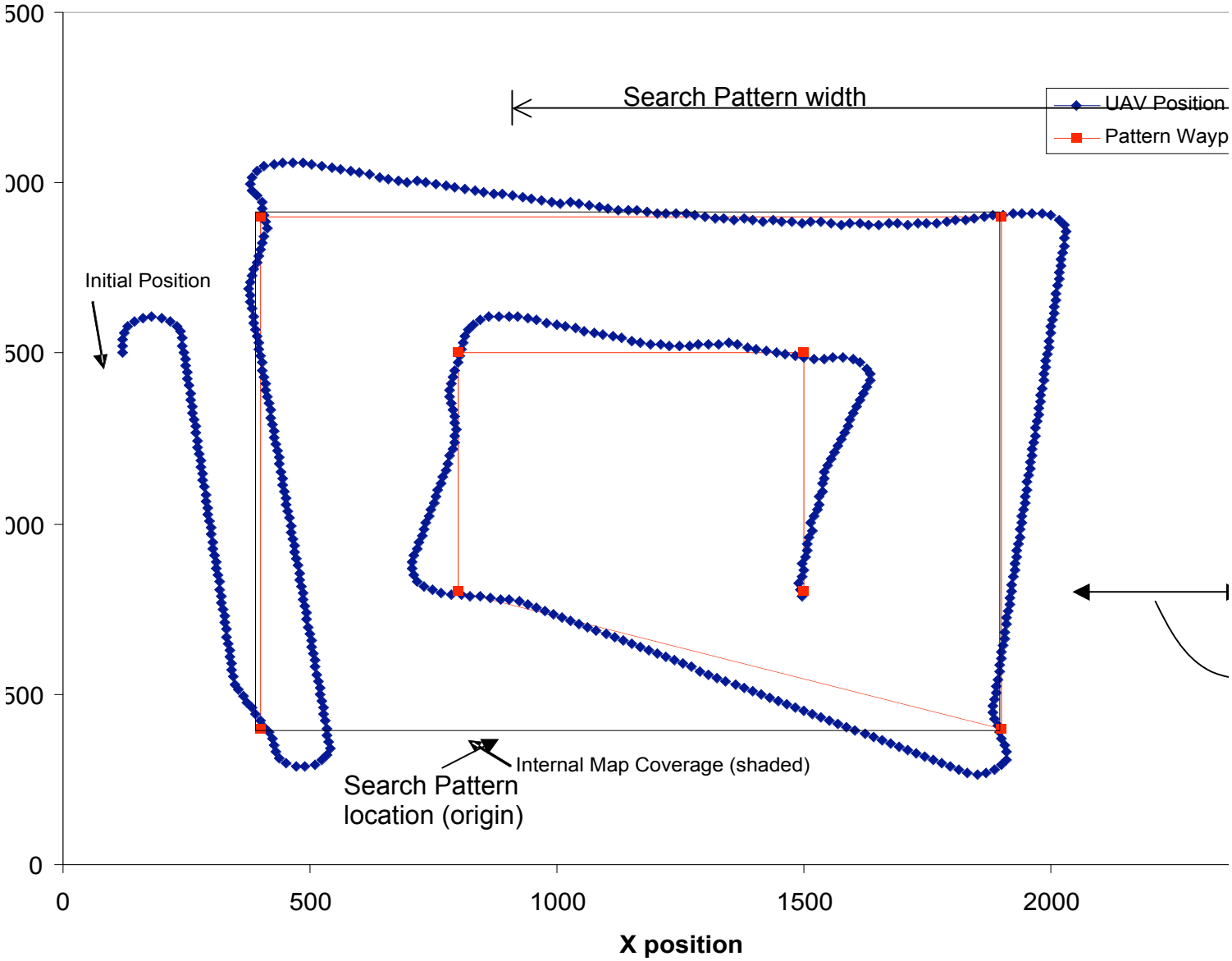
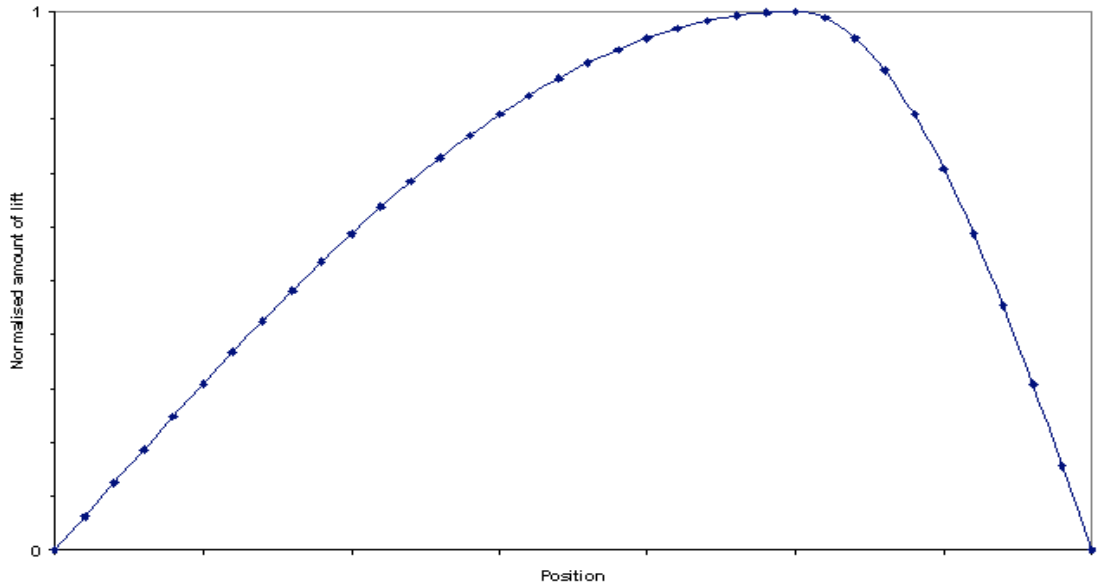
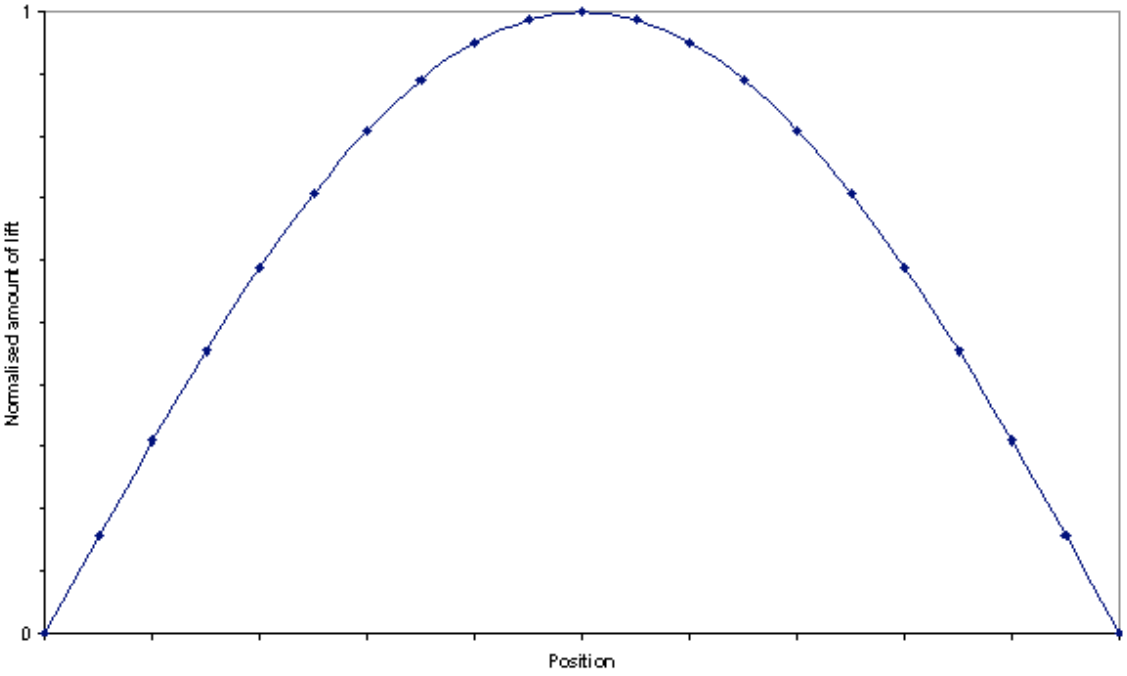


Figure 1. Path of the UAV following the search plan.



(a)



(b)

Figure 1.(a),(b)Cross-sections of simple ridge and thermal lift amounts respectively.

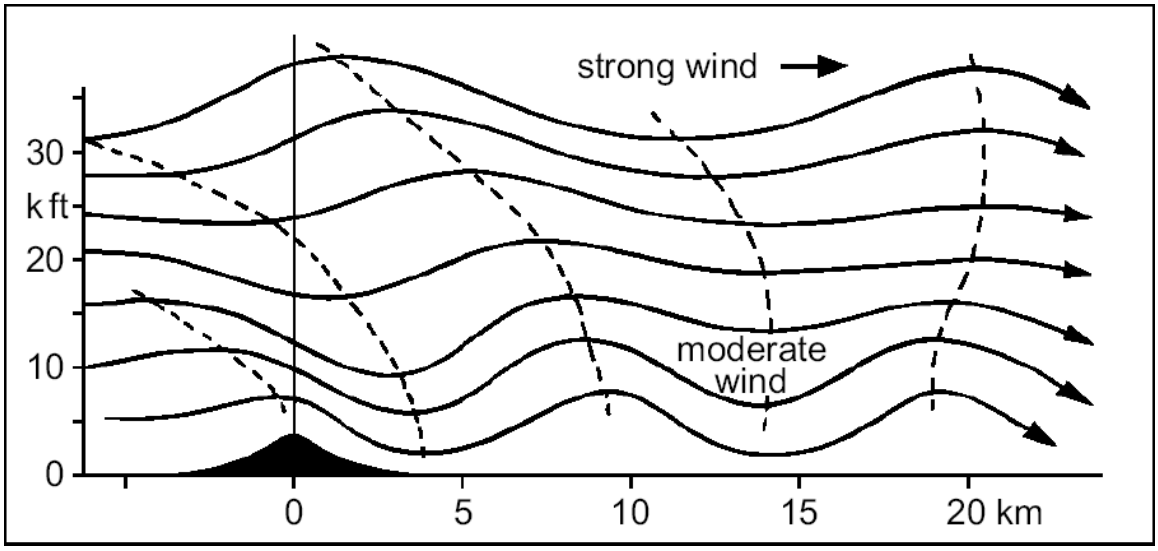


Figure 1. Example airflows over a ridge.

Figure 1, reproduced from [5], shows the paths taken as air travels over a ridge. As can be seen, attempting to create a mathematical model for this behaviour is worthy of its own thesis project!

In order to model this behaviour, the following simplifications were made:

1. At a certain elevation above a ridge, the effect of the ridge becomes negligible. The elevation at which this occurred was termed the “neutral elevation”.
2. The velocity of the wind is constant with respect to altitude. In actual fact, air travels more slowly along the ground than above it.

Figure 1 shows the cross-sectional view of the proposed ridge. It can be seen how the ridge deflects the air at lower altitudes most severely, whilst the air nearer the neutral elevation is less affected.

Along the plateau of the ridge, the airstreams show that the density of air has increased – in the figure it approximately triples. This will have an effect on the speed at which the air travels. If we consider the ridge as analogous to a narrowing in a pipe, the effect can be determined. When a unit volume of air enters the wide end of a pipe, it must displace the same volume of air out the narrow end. The equation of continuity[6] states:

$$\Delta Volume = A_1 v_1 \Delta t = A_2 v_2 \Delta t = 0$$

$$\therefore A_1 v_1 = A_2 v_2$$

Thus the speed at which the air travels increases as the cross-sectional area of the pipe decreases. The same applies for the ridge. As the same volume of air is compressed as it traverses the plateau, its speed increases as a ratio of the neutral elevation and the plateau height.

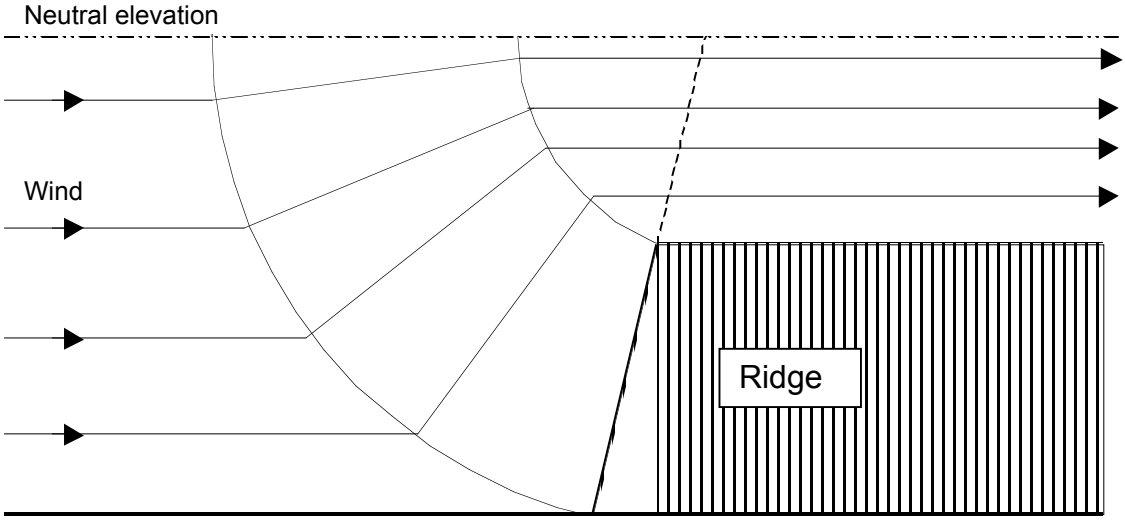


Figure 1. Cross-section of ridge showing streamlines of air at differing altitudes.

As the air is transitioning from unaffected (before the ridge) to being uniformly compressed by the plateau, it passes through non-uniform compression which has been modelled as linear. That is, the amount of compression (and hence increase in speed) changes linearly from uncompressed to fully compressed.

The model of the ridge includes a configurable valley. The air flows over the ridge, is compressed along the plateau and then spreads out again in the valley. The effect can be seen in Figure 1, if you consider the wind as travelling in the opposite direction.

Using the advanced ridge model, the user can specify the position and parameters of any number of ridges they wish to include in the terrain. The program reads the configuration file and successively adds each ridge into the otherwise flat terrain. The terrain is divided into a grid of cells, with each cell being further subdivided. The vertices of the subdivision store height information of the terrain for that point. The division into cells allows the program to determine which cells are visible on the display and which aren't. By identifying only those cells which can be seen, the processing power required to render a scene to the display is greatly reduced. The process of selecting the visible cells is called "culling". The particular method used is called "frustum-tested culling" and was written by [7].

2.2.2. Wind Model

The wind used for the simulation has been modelled as having a constant direction and a variable strength. The average strength and the maximum percentage change of the wind are specified as part of the simulation. These parameters are set in the terrain description file, allowing them to be changed without recompilation of the program.

2.2.3. GPS Acquisition Delay

GPS units by their very nature do not provide an instantaneous measurement of their position. When asked to give their position, there is a delay whilst the units interrogates the signals from the satellites and triangulates its location. If the GPS unit is moved during this time (i.e. the UAV is moved), the result will be correct for the old position - not the new. This means the GPS is continually playing “catch up” on the actual position of the UAV. A typical value for the acquisition delay is 1 to 1.5 seconds.

Because of the GPS acquisition delay, the UAV never really knows where it is. This can cause trouble when the UAV pilot must make decision based upon position. Examples of when problems may occur are:

- determining when a waypoint was been reached;
- deciding how hard to turn to reach a waypoint; and,
- storing and retrieving information based upon position.

The third point is of particular interest for this project. Because of the delay, lift information for the current location is stored at the position of the UAV *GPS_Acquisition_Delay* seconds ago. When the UAV is travelling at a cruise speed of 20 m/s, this can mean storing information up to 30m away from its correct location.

The effect of acquisition delay is achieved in the program by maintaining a list of position-time pairs. When a position request is received, the list is scanned for the pair whose time value is the closest to being *GPS_Acquisition_Delay* seconds less than the current time.

If the direction of travel of the UAV was constant and there was no wind, the acquisition delay and its accompanying position error could be overcome by simply using an offset. The problem becomes more difficult when the UAV is allowed travel in any direction. A possible solution is to calculate how much the UAV has moved during the acquisition period and use this to determine the UAV’s real position – a kind of “dead reckoning” approach. The addition of wind will introduce an offset, whose direction and magnitude will be controlled by the direction and strength of the wind. If, for example, the wind is travelling northward at $5ms^{-1}$ and the acquisition delay is 1.5 seconds, an offset of 7.5m to the north would be induced. The error caused by wind does not accumulate as the wind only affects the position determination over the acquisition period.

The solution taken in this project to overcome the effect of GPS acquisition delay is to make the lift seeking algorithms dynamic enough so that the effect of the imprecise location is negligible.

3. Results & Discussion

It is difficult to generate an absolute measure for comparing the performance of the advice system versus an unaided system, as so much depends upon the location of the UAV, the lift and the map. Nonetheless, an attempt to obtain reasonable results will be made. The metrics chosen to enable measurement and comparison are:

- Average amount of lift per unit time ($m.s^{-1}$); and,
- Fraction of time spent in lift (s / s).

The same terrain structure was used to generate all results. The configuration file for the terrain, known as the "Bendy Ridge", is listed in Appendix 8.3, whilst a top-down view is provided in **Error! Reference source not found.**.

The results were obtained by allowing the simulation to run for ten minutes using a half-second time step. The ten minutes begins once the advice system begins advising. Also,

- The UAV starts from position (500, 500) and is heading north.
- The UAV is limited to fly within the altitudes of 750 and 1200 metres (this prevents the UAV from flying into the ridge or above the lift).
- The wind speed is $7m.s^{-1}$ and is coming at an angle of 20° east of north.

Each test is run three times and the results are averaged.

3.1. *Pre-advice Pattern*

The first test is to determine the benefit of using the Preadvice Pattern Flight (see 2.1.3) and which grid width performed the best. All simulations used the following parameters:

- The map is located at (1500, 1000);
- The map length is 1000 metres; and,
- The distance between map elements is 20m.

The results of the tests are summarised in Table 1, below. It shows that using the pre-advice flight pattern provides a significant improvement to the quality of the advice the system can give. This is evidenced by the dramatic improvement in average lift per unit time. The results confirm the expectation that providing the advice system with initial information allows it to perform better.

It can also be seen that the fraction of time spent in lift decreases as the pattern width decreases. The smaller the pattern width, the longer it takes the UAV to fly the pattern. Using this, the (human) UAV pilot can make a trade-off between the fraction of time spent in lift and how quickly the UAV begins giving advice.

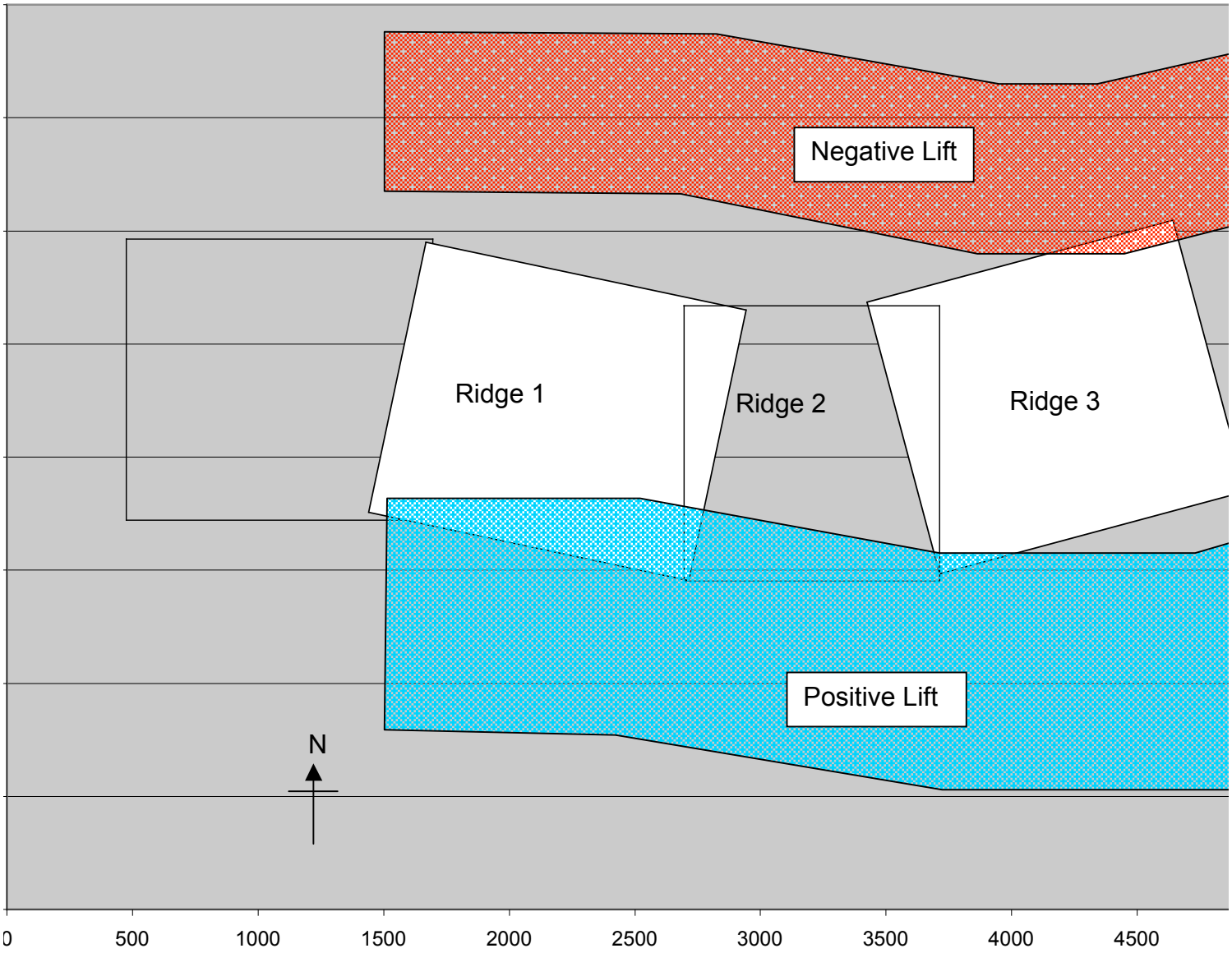


Figure 1. A top-down view of the "Bendy Ridge".

Table 1. Parameters and results of test A.

Test Number	Pattern Width (m)	Average lift / unit time (m/s)	Fraction of time in lift (s/s)
A1	Not Used	-0.24	0.14
A2	50	0.55	0.39
A3	100	0.56	0.38
A4	200	0.55	0.34
A5	300	0.49	0.25
A6	500	0.52	0.27
A7	700	0.36	0.18

3.2. *Advice Performance*

The next test is to compare the performance of a UAV with the advice system against an unaided one.

Firstly, we calculate the value of the metrics for an unaided system. To do so, we use reasoning and the following assumptions:

- The UAV is in an area with lift but that the location of the lift is unknown; and,
- The UAV adopts a strategy of flying a fixed-location circle.

The chance of the UAV flying in lift will depend on how "lift rich" the area is. For a long, straight ridge such as a sand dune, there is positive lift only at the windward face and we can divide the area in three parts – two no-lift areas with a lift area in between. If the no-lift areas are twice as long as the lift areas, the fraction of time spent in lift will be ~0.20. This number will decrease as the lift section becomes thinner. In the best lift, the vertical air velocity will be $\cos(20°) \cdot 7 = 6.6 m/s$ directly upward. In the worst case, the lift will be zero. Thus, the average vertical air velocity will be $(\frac{6.6+0}{2}) = 3.3 m/s$ upward. The average amount of lift per unit time is therefore $(3.3) \cdot (0.20) \approx 0.66 m/s$.

These figures are determined using ideal conditions. The completely vertical air will only be found at the very face of the ridge which is a very dangerous place to fly – a place the Advice System does not fly. As such, the unaided UAV's performance will be over-represented by these figures.

If instead a mountain-type ridge is used, there are two lift areas and two no-lift areas. One of the lift areas would be positive, the other negative. Thus, the fraction of time spent in lift would be ~0.5 and the average amount of lift per unit time would be ~0. If there is a plateau, the fraction of time spent in lift would decrease.

Table 2 shows the parameters used when performing test B. Table 3 shows the results of the test. We can now compare the advice system against an unaided one.

Table 2. Parameters of test B.

Test Number	Map Position (m, m)	Map Length (m)	Map Separation (m)	Pattern Width (m)
B1	(1500, 1000)	1000	20	200
B2		1000	40	
B3		1520	20	
B4		1520	40	
B5		2000	40	
B6	(1500, 500)	1000	20	Not used
B7				200

Table 3. Results of test B – the performance of aided and unaided UAVs.

System	Average lift / unit time (m/s)	Fraction of time in lift (s/s)
Unaided		
Sand Dune-type	~0.66	~0.20
Mountain-type	~0.0	~0.50
Average (if types equally likely)	*~0.33*	*~0.35*
Aided		
B1	0.53	0.32
B2	0.53	0.32
B3	0.36	0.22
B4	0.39	0.24
B5	0.14	0.29
Average	*0.39*	*0.28*
B6	0.67	0.19
B7	0.53	0.34
Average	*0.60*	*0.27*
Aided Average	**0.50**	**0.28**

Tests B1 through B5 are testing the system for a variety of map length and elements separations to see how they compared. The results show that the change in separation from 20m to 40m made little difference to the fraction of time spent in lift. The change in map size, however, did have an effect. The smaller map size has a larger average lift. This may be due to the larger map covering more area and causing the UAV to travel in negative lift areas when flying the Pre-advice Pattern.

Test B6 & B7 are conducted in a more lift rich area. They once again confirm that the performance of the UAV is improved by using the Pre-advice Pattern.
Overall, the advice system performs slightly better in the average lift per unit time and slightly worse in the fraction of time spent in lift. This

leads me to believe that the algorithm used in the advice system needs revisiting, in order to improve its performance.

I would not say the project is a failure. Although the performance of the advice system is not outstanding, the framework is in place to allow new algorithms to be easily and rapidly developed and tested.

4. Time Analysis

ID	Task Name	Duration
1	Develop simulation models	1 wk
2	Develop & test Advanced Turn algortihm	4 wks
3	Consider, implement& test second-round effects	6 wks
5	Write thesis	4 wks
6	Write conference paper	2 wks
7	Design poster	2 wks
14	Submit Design Document	0 days
15	Submit poster	0 days
16	Submit draft thesis	0 days
17	Submit Thesis	0 days

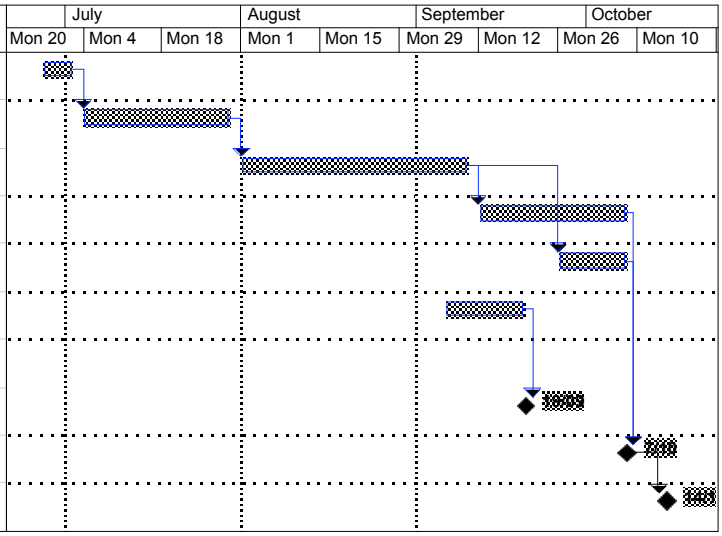


Figure 1. Proposed project timeline, as at 5 May, 2005.

A timeline for the planned progression of the project was created in May and can be seen in Figure 1. Actual development of the project proceeded in the indicated order but each task did not necessarily begin and finish on the dates specified. In particular, it took longer than expected to finish the simulation models. Even though the timeline was not strictly adhered to, it did proove valuable for breaking down and ordering the tasks needed to complete the project.

5. Extensions and Improvements

In a simulation project such as this, work could continue indefinitely making improvements. Some of these improvements, and possible extensions, include:

- In the wind model, the wind speed increases as the airflow is compressed over the ridge. Currently, there is a sharp change of wind speed above and below the neutral elevation. This effect should be investigated to determine if it is a suitable approximation or whether a change needs to be made.
- The wind model has no rotor.

6. Conclusion

7. References

[1] Various, "Flightgear," vol. 2005, pp. An open source flight simulator.

[2] D. Chapman, "Sss - Slope Soaring Simulator," vol. 2005.

[3] D. Sunday, "About Lines and Distance of a Point to a Line (2d & 3d)," vol. 2005.

[4] T. Bradbury, "Wave Soaring over the British Isles (Some Theoretical Aspects and Practical Observations)," vol. 2005. online, 1984.

[5] T. Bradbury, "A Look at Wave Clouds," *Free Flight*, 1992.

[6] Halliday, Resnick, and Walker, *Fundamentals of Physics*, vol. 2, Sixth ed: John Wiley & Sons, Inc., 2001.

[7] M. Morley, "Markmorley.Com | Frustum Culling in Opengl," 2000.

8. Appendices

8.1. *Description of Configuration File*

8.2. *Program Control*

8.3. *"Bendy Ridge" Configuration File*

```
# Terrain map for UAV simulator
# Each line is read separately, and generally looks like (without the
'#'):
#
# <attribute> <value>
#
# '#' is comment character - anything following a '#' gets ignored.
# When things are specified more than once, it is the first line
# that counts.
#
# Case is important.

### Wind setting
WINDSpeed           7       # metres per second
WINDAngle           30      # compass point wind blows towards
WINDFluctuation     0.1     # percentage that the wind speed may
randomly fluctuate by

### UAV flight settings
UAVFloor            750     # UAV will try not to go below this height
(metres).
UAVCeiling          1200    # UAV will try to stay below this
(metres).
UAVStartX           100     # Start position, x component (metres).
UAVStartY           1500    # y component (metres).
UAVStartZ           700     # initial height (metres).
UAVGPSDelay         1.2     # delay in acquisition of the GPS
(seconds).
UAVStartHeading     110     # initial heading (degrees).

## UAV Internal Map Settings
MAPWidth            50      # Number of columns in map
MAPHeight           50      # Number of rows in map
MAPSeparation       20      # Distance in metres between adjacent rows
& columns
MAPLongitude        500 # X axis position of the map (metres)
MAPLatitude         500 # Y axis position of the map (metres)
MAPSearchWidth      500     # Distance between flight paths in the
search pattern
MAPSearchEnabled    true    # Use the search pattern? (true/false)
MAPEscapeDistance    2000   # Distance from centre of map before UAV
automatically heads back towards centre. Use zero to disable.

####################
###   TERRAIN   ###
# number of ridges in map
num_ridges      4
# number of interpolation flats
num_flats       1

## RIDGE 1
```

```
begin ridge1
#ridge position
ridge_origin_x     500     # (metres)
ridge_origin_y     1700    # (metres)
ridge_angle        0       # integer - angle measured in degrees CCW
from x-axis
ridge_length       1200    # (metres)
left_smoothen_percent  0.2     # fraction, smooths the left edge of
the ridge to the ground
right_smoothen_percent 0.0     # fraction, smooths the right edge of
the ridge to the ground

#ridge geometry
rise_angle         60      #degrees
fall_angle         40      #degrees
plateau_elevation  500     #metres
valley_elevation   300     #metres
plateau_length     800     #metres
neutral_elevation  1300        #metres

## RIDGE 2
begin ridge2
ridge_origin_x     1512    #all integers
ridge_origin_y     1750
ridge_angle        345     # integer - angle measured CCW from x-axis
ridge_length       1147
left_smoothen_percent  0.0     # fraction, smooths the left edge of
the ridge to the ground
right_smoothen_percent 0.0     # fraction, smooths the right edge of
the ridge to the ground

rise_angle         60      #degrees
fall_angle         40      #degrees
plateau_elevation  600     #metres
valley_elevation   300     #metres
plateau_length     724     #metres
neutral_elevation  1300        #metres

## RIDGE 3
begin ridge3
ridge_origin_x     2700    #all integers
ridge_origin_y     1442
ridge_angle        0       # integer - angle measured CCW from x-axis
ridge_length       1000
left_smoothen_percent  0.0     # fraction, smooths the left edge of
the ridge to the ground
right_smoothen_percent 0.0     # fraction, smooths the right edge of
the ridge to the ground

rise_angle         60      #degrees
fall_angle         40      #degrees
plateau_elevation  500     #metres
valley_elevation   300     #metres
plateau_length     700     #metres
neutral_elevation  1300        #metres

## RIDGE 4
begin ridge4
ridge_origin_x     3700    #all integers
ridge_origin_y     1442
ridge_angle        15      # integer - angle measured CCW from x-axis
```

```
ridge_length       1300
left_smoothen_percent  0.0     # fraction, smooths the left edge of
the ridge to the ground
right_smoothen_percent 0.15        # fraction, smooths the right edge
of the ridge to the ground

rise_angle         60      #degrees
fall_angle         40      #degrees
plateau_elevation  400     #metres
valley_elevation   300     #metres
plateau_length     700     #metres
neutral_elevation  1300        #metres


begin flat1
#flat vertices - defined clockwise, integers
x1          1400
y1          3200

x2          1400

y2          6000

x3          3000
y3          6000

x4          2000
y4          3200
```

8.4. *Source Code*

Advice.h

```
/*-------------------------------------------------------------------------
File name:       Advice.h
Author:          Ashley Goodwin
Created:     24 August 2005
-------------------------------------------------------------------------
         The Advice header file specifies the parameters that the Turn Advice
         system will use to determine its advice. They are placed in a seperate
         header to prevent unneccessary recompilation of other non-path files.
-------------------------------------------------------------------------*/
#ifndef _ADVICE_H_
#define _ADVICE_H_

#define USE_CIRCLE_ADVICE

//------------------+---------------------------------------+----------------\\
//                      |       Advanced Advice parameters     |                \\
//------------------+---------------------------------------+----------------\\
//                                  Circle parameters                          \\
//
const int Circle_NumAngles = 6;      // num angles to try in quadrant

const int Circle_SearchDist = 2500;     // metres from UAV to search radially

#endif
```

Camera.cpp

```
#include <math.h>
#include "camera.h"

// Position the cemare at an arbitrary point.
void Camera_t::PositionCamera(float pos_x,  float pos_y,  float pos_z,
                                  float view_x, float view_y, float view_z,
                                  float up_x,   float up_y,   float up_z)
{
         mPos    = Vector3D(pos_x,  pos_y,  pos_z ); // set position
         mView   = Vector3D(view_x, view_y, view_z); // set view
         mUp     = Vector3D(up_x,   up_y,   up_z  ); // set the up vector
}
// Move the camera in the direction of sight.
void Camera_t::MoveCamera(float speed)
{
         Vector3D vVector = mView - mPos;   // Get the new vector

         mPos.x  = mPos.x  + vVector.x * speed;
         mPos.y  = mPos.y  + vVector.y * speed;
         mView.x = mView.x + vVector.x * speed;
         mView.y = mView.y + vVector.y * speed;
}
// Rotate the view around the current position
void Camera_t::RotateView(float speed)
{
         Vector3D vVector = mView - mPos;   // Get the view vector

         mView.y = (float)(mPos.y + sin(speed)*vVector.x + cos(speed)*vVector.y);
         mView.x = (float)(mPos.x + cos(speed)*vVector.x - sin(speed)*vVector.y);
}
// Rotate around the viewpoint
```

```
void Camera_t::RotatePosition(float speed)
{
          Vector3D vVector = mPos - mView;

          mPos.z = (float)(mView.z + sin(speed)*vVector.x + cos(speed)*vVector.z);
          mPos.x = (float)(mView.x + cos(speed)*vVector.x - sin(speed)*vVector.z);
}
// Look up/down.
void Camera_t::AngleView(float speed)
{
          mView.z += speed;
}
```

Camera.h

```
#ifndef _CAMERA_H_
#define _CEMARA_H_
#include "vector.h"      // This file requires vectors

#define CAMERASPEED  0.2f               // The Camera Speed
//-------------------------------------------------------------
// The Camera class declaration
class Camera_t
{
          public:

              Vector3D mPos;
              Vector3D mView;
              Vector3D mUp;

              // This function makes it possible to rotate around a given point
              void RotatePosition(float speed);
              void MoveCamera(float speed);
              void RotateView(float speed);
              void AngleView(float speed);
              void PositionCamera(float pos_x, float pos_y,float pos_z,
                                  float view_x, float view_y, float view_z,
                                  float up_x,   float up_y,   float up_z);
};

#endif
```

Config_File.cpp

```
/*!
  Sss - a slope soaring simulater.
  Copyright (C) 2002 Danny Chapman - flight@rowlhouse.freeserve.co.uk

  \file config_file.cpp
*/
#include <new>
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;

#include "config_file.h"
#include "trace.h"

const string Config_file::comment_char("#");
```

```
const string Config_file::delims(" \t\r");
const string Config_file::digits("1234567890.-");

void Config_file::reset(ifstream * & stream)
{
  delete stream;
  stream = new ifstream(m_file_name.c_str());
}

Config_file::Config_file(const string &file_name, bool & success)
  :
  m_file_name(file_name),
  m_seq_file_stream (new ifstream( m_file_name.c_str() )  ),
  m_rand_file_stream(new ifstream( m_file_name.c_str() )  )
{
          if( !m_rand_file_stream->is_open() || !m_seq_file_stream->is_open() )
          {
              success = false;
              return;
          }
          success = true;
}

/*!  Copy constructor - ensures that the new streams have the same
  offsets as the original, but they are new streams */
Config_file::Config_file(const Config_file & orig)
          :
          m_file_name(orig.m_file_name),
           m_seq_file_stream(new ifstream(m_file_name.c_str())),
          m_rand_file_stream(new ifstream(m_file_name.c_str()))
{
          // ensure that the streams are in the same position as the originals
          streampos pos = orig.m_seq_file_stream->tellg();
          m_seq_file_stream->seekg(pos);
          pos = orig.m_rand_file_stream->tellg();
          m_rand_file_stream->seekg(pos);
}

/*! ensures that the new streams have the same
  offsets as the original. */
Config_file Config_file::operator=(const Config_file & orig)
{
          return Config_file(orig);
}

Config_file::~Config_file()
{
          if (m_seq_file_stream)
              delete m_seq_file_stream;
          if (m_rand_file_stream)
              delete m_rand_file_stream;
}

bool Config_file::get_next_processed_line(ifstream * stream,
                                          vector<string> & words)
{
          if (!stream)
          {
              TRACE("input file is not open: " <<  m_file_name);
              return false;
          }
```

```
char char_line[1024];
while (stream->getline(char_line, 255))
{
    string::size_type beg_index, end_index;
    string line(char_line);
    words.clear();

//      cout <<
"============================================================\n";
//      cout << line << endl;

    // remove comment
    beg_index = line.find(comment_char);
    if (beg_index !=  string::npos)
    {
    line.resize(beg_index);
//      cout << "After removing comment: " << line << endl;
    }

    beg_index = line.find_first_not_of(delims);


    while (beg_index != string::npos)
    {
    end_index = line.find_first_of(delims, beg_index);

    if (end_index == string::npos)
    {
        // end of word is end of line
        end_index = line.length();
    }

//      cout << "[" << beg_index << "," << end_index << "]";
    string a_word(line.substr(beg_index, end_index-beg_index));
//      cout << a_word.length() << " @" << a_word << "@" << endl;

    // now we have a word

    // convert true/false into 1/0
    if ( (a_word == "true") ||
        (a_word == "True") ||
        (a_word == "TRUE") )
    {
        a_word = "1";
    }
    else if ( (a_word == "false") ||
                (a_word == "False") ||
                (a_word == "FALSE") )
    {
        a_word = "0";
    }


    words.push_back(a_word);

    // find the next word
    beg_index = line.find_first_not_of(delims, end_index);
    }

    if (words.size() > 0)
```

```
            {
        //        cout << "Individual words: @";
        //        // output the words
        //        for (unsigned int i = 0 ; i < words.size() ; ++i)
        //        {
        //          cout << words[i] << "@";
        //        }
        //        cout  << endl;

            // found our words - now go home!
            return true;
            }

            // words is empty - try again
        }

        // no more lines
        return false;
}


Config_file::Config_attr_value Config_file::get_next_attr_value()
{
        Config_attr_value attr_value;

        while (true)
        {
            vector<string> words;
            if (false == get_next_processed_line(m_seq_file_stream, words))
            {
                //reached end of file
                return attr_value;
            }
            // we need at least one attribute and one value to do anything
            if (words.size() > 1)
            {
                // the easy bit
                attr_value.attr = words[0];

                // work out the type. We assume that all the values have the
                // same type, so just check the first.
                if ( (words[1] == "true") || (words[1] == "TRUE") )
                {
                    attr_value.value_type = Config_attr_value::BOOL;
                }
                else if ( (words[1] == "false") || (words[1] == "FALSE") )
                {
                    attr_value.value_type = Config_attr_value::BOOL;
                }
                else if (words[1].find_first_not_of(digits) != string::npos)
                {
                    attr_value.value_type = Config_attr_value::STRING;
                }
                else
                {
                    attr_value.value_type = Config_attr_value::FLOAT;
                }

                unsigned int i;
                for (i = 1 ; i < words.size() ; ++i)
                {
```

```
                    Config_attr_value::Value val;
                    switch ( attr_value.value_type )
                    {
                    case Config_attr_value::BOOL:
                        val.bool_val = ( (words[i] == "true") || (words[i] == "TRUE")
        );
                        break;

                    case Config_attr_value::FLOAT:
                        val.float_val = (float) atof(words[i].c_str());
                        break;

                    case Config_attr_value::STRING:
                        val.string_val = words[i];
                        break;

                    default:
                        break;
                    }

                    ++attr_value.num;
                    attr_value.values.push_back(val);
                }
                // can return
                return attr_value;
            }
            // words.size() == 0, so have another go
        }
}


Config_file::Config_attr_value Config_file::find_next_attr_value(
  const string & attr)
{
        Config_attr_value attr_value;

        while (true)
        {
            attr_value = get_next_attr_value();

            // have we reached the end of file?
            if (attr_value.num == 0)
                return attr_value;

            if (attr_value.attr == attr)
            {
            return attr_value;
            }
            // no luck yet - try next line
        }
        return attr_value;
}

bool Config_file::find_new_block(const string & block_name)
{
        Config_attr_value attr_value;

        while (true)
        {
            attr_value = find_next_attr_value("begin");
            if (Config_attr_value::INVALID == attr_value.value_type)
```

```
            {
            TRACE("Config_file::find_new_block() End of config file reached");
            return false;
            }
            if (Config_attr_value::STRING == attr_value.value_type)
            {
            if (block_name == attr_value.values[0].string_val)
            {
                TRACE("Config_file::find_new_block() Found block");
                return true;
            }
            }
        }
        // keep compiler happy
        return false;

}
```

Config_File.h

```
/*
  Sss - a slope soaring simulater.
  Copyright (C) 2002 Danny Chapman - flight@rowlhouse.freeserve.co.uk
*/
#ifndef _CONFIG_FILE_H_
#define _CONFIG_FILE_H_

#include <string>
#include <sstream>
#include <fstream>

#pragma warning (disable: 4786)

#include <vector>
using namespace std;

#include "trace.h"

//! Utility class for opening and parsing text config files
/*!

  Note that there are two ways of parsing configuration files:
  <ol>
  <li> Specifying an attribute that you suspect exists in the config
  file, and obtaining the value(s) associated with it.
  <li> Asking for the next attribute/value pair in the file.
  </ol>
  It is possible to mix-and-match the two methods with this class.

  The comment character in the config file is assumed to be "#".
*/
class Config_file
{
public:
        float SingleFloat( string s )
        {
            float r = 0;
            get_value( s, r );
            return r;
        }
```

```
        bool SingleBool( string s )
        {
            bool r = false;
            get_value(s, r);
            return r;
        }
        long SingleLong( string s )
        {
            return (long) SingleFloat( s );
        }

  //! success indicate ability to open file_name
  Config_file(const string & file_name, bool & success);
  //! Copy constructor
  Config_file(const Config_file & orig);
  //! Equals operator
  Config_file operator=(const Config_file & orig);

  //! Destructor closes any open file
  ~Config_file();

  //! Indicates the config file name
  string get_file_name() const {return m_file_name;}

  //! Function for reading a single value.
/*!  This returns the value for the first occurence of attr in the
  file. It does not affect the location used in the get_next
  functions.

  If the get fails, the value (passed by reference) is unmodified.

  \return true if successful. False if either the attribute isn't
  found, or it has no value associated with it. In the case that
  multiple values are found, only the first is used (an error message
  is generated). */
  template<class T>
  bool get_value(const string & attr, T & value)
    {
      vector<T> values;
      bool retval = get_values(attr, values);
      if (false == retval || values.size() == 0)
      {
                LOG("Couldn't find '" << attr << "' in settings.");
        return false;
      }
      if (values.size() > 1)
      {
                LOG("Multiple values for attribute '" << attr << "'. Using first
          found.");
      }
      value = values[0];
      return true;
    }


  //! Version that asserts on failure
//! same as get_value, except asserts in case of failure
  template<class T>
  void get_value_assert(const string & attr, T & value)
    {
      bool retval = get_value(attr, value);
```

```
      if (retval != true)
      {
        TRACE("Failed to find attribute " << attr);
      }
    }


  //! Get multiple values for an attribute
/*!  This returns the values for the first occurence of attr in the
  file. It does not affect the location used in the get_next
  functions.

  If the get fails, the values (passed by reference) is unmodified.

  \return true if successful. False if the attribute is not found, or
  if it has no values associated with it.*/
  template<class T>
  bool get_values(const string & attr, vector<T> & values)
    {
      // reset the random access stream
      reset(m_rand_file_stream);

      vector<string> words;

      while (true == get_next_processed_line(m_rand_file_stream, words))
      {
        if ( (words.size() > 0) && (words[0] == attr) )
        {
          // found it!
          values.clear();
          if (1 == words.size())
          {
            TRACE("Found attribute " << attr << " but no value");
            return false;
          }
          // we don't actually know the type of value, so use its ability
          // to read in from a stream.
          unsigned int i;
          string everything;
          for (i = 1 ; i < words.size() ; ++i)
          {
            everything += words[i];
            everything += " ";
          }
          istringstream is(everything);

          T value;
//      cout << "Read: " << attr;
          for (i = 1 ; i < words.size() ; ++i)
          {
            is >> value;
            values.push_back(value);
//        cout << value << " ";
          }
//      cout << endl;
          return true;
        }
      }
      // didn't find what we were looking for
      return false;
    }
```

```
//! Contains the details of an attr/value configuration entry. Needs
//! to be able to cope with multiple values, of an unknown type.
struct Config_attr_value
{
  Config_attr_value() : attr("None"), value_type(INVALID), num(0) {};
  string attr; //!< The attribute name
  //! INVALID is used if nothing found
  enum Value_type {INVALID, BOOL, FLOAT, STRING} value_type;
  unsigned int num; //!< Number of values - 0 if not found
  struct Value
  {
    bool   bool_val;
    float  float_val;
    string string_val;
  };
  vector<Value> values;
};
//! Returns the next attr/value entry. In the case of EOF, sets the
//! type = INVALID, and num = 0
Config_attr_value get_next_attr_value();

//! Returns the next line of values as a vector (i.e. as in
//! Config_attr_value but with no attr).
template<class T>
bool get_next_values(vector<T> & values, int num = 0)
  {
    values.clear();
    vector<string> words;

    if (false == get_next_processed_line(m_seq_file_stream, words))
    {
      // reached end of file
      return false;
    }
    T value;
    if (num > 0)
    {
      if ((int) words.size() < num)
      {
        TRACE("expected " << num << " values, found " <<  words.size());
              cerr << "values read in were:" << endl;
        unsigned i;
        for (i = 0 ; i < words.size() ; ++i)
        {
          TRACE(words[i] );
        }
      }
      else if ((int) words.size() > num)
      {
        TRACE("expected " << words.size() << " values, found " << num << " so
       ignoring the extras");
        words.resize(num);
      }
    }
    for (unsigned i = 0 ; i < words.size() ; ++i)
    {
      istringstream is(words[i]);
      is >> value;
      values.push_back(value);
    }
```

```
      return true;
    }


  //! Gets the next value, asserting that it is of the specified
  //! attribute
  template<class T>
  void get_next_value_assert(const string & attr, T & value)
    {
      vector<string> words;

      while (true == get_next_processed_line(m_seq_file_stream, words))
      {
        if (words.size() < 2)
        {
          TRACE("Expecting at least two words with attribute " << attr);
        }
        if (words[0] != attr)
        {
          TRACE("Expected attribute " << attr << ": found " << words[0]);
        }

        // we don't actually know the type of value, so use its ability
        // to read in from a stream.
        istringstream is(words[1]);
//    cout << "Attribute: " << attr;
        is >> value;
//    cout << value << endl;
        return;
      }
      TRACE("Attribute " << attr << " not found");
    }


  //! Gets the next value, WITHOUT asserting that it is of the specified
  //! attribute. value is unchanged if the attr is not as requested,
  //! and the read position does not advance (except for white-space).
  //! the return value indicates if the read was successful.
  template<class T>
  bool get_next_value(const string & attr, T & value)
    {
      vector<string> words;
      // store position

      // Apparantly streampos should really be std::ios::pos_type, but
      // maybe my compiler is a bit old-fashioned?
      streampos pos = m_seq_file_stream->tellg();

      while (true == get_next_processed_line(m_seq_file_stream, words))
      {
        if (words[0] == attr)
        {
          // we don't actually know the type of value, so use its ability
          // to read in from a stream.
          istringstream is(words[1]);
//      cout << "Attribute: " << attr;
          is >> value;
//      cout << value << endl;
          return true;
        }
        else
```

```
        {
          // we have to rewind
          m_seq_file_stream->seekg(pos);
          return false;

        }
      }
      TRACE("Attribute " << attr << " not found");
      return false;
    }


  /// finds the next block starting with "begin block_name". Returns
  /// true if found, false if not.
  bool find_new_block(const string & block_name);

  //! Finds the next occurence of the desired attribute, and returns
  //! the attr/value result
  Config_attr_value find_next_attr_value(const string & attr);

  /*!  resets the internal sequential file descriptor to point to the
    beginning of the file - so the next call to get_next_attr_value
    will return the first.  */
  void reset() {reset(m_seq_file_stream);}

  //! Returns the stream pointer used in get_next_attr_value(...).
  ifstream * get_stream() const {return m_seq_file_stream;}

private:
  //! Seeks the specified stream to the beginning of file
  void reset(ifstream * & stream);

  //! gets a vector of words for the next line. Returns false at EOF
  bool get_next_processed_line(ifstream * stream, vector<string> & words);

  const string m_file_name;
  ifstream *m_seq_file_stream;  //!< stream used for sequential access
  ifstream *m_rand_file_stream; //!< stream used for random access

  // some static consts
  static const string comment_char; //!< Comment character
  static const string delims;       //!< "word" delimiters
  static const string digits;       //!< used to identify numbers

};

#endif
```

Control.cpp

```
/*-----------------------------------------------------------------------------
         The main.cpp file contains the autipilot system.
-----------------------------------------------------------------------------
File name:       main.cpp
Author:          Ashley Goodwin
Created:     5 March, 2005
-----------------------------------------------------------------------------*/
#include <string>
#include <iostream>
#include <math.h>
```

```
using namespace std;

#include "trace.h"
#include "UAV.h"
#include "sim_map.h"
#include "map.h"
#include "path.h"
#include "advice.h"
#include "GPSList_t.h"

Pattern_t        Pattern;
Map_t               map(0,0,0,0,0,0);
GPSList_t        gpl;
extern Terrain_t terrain;
extern float     world_time;

bool InitPilot( UAV_t *plane, Map_t * pmap, Vector3D start, float heading, float
          airspeed,
                    long floor, long ceiling, long gps, float MapSearchWidth, bool
          MapSearchEnabled)
{
          using namespace std;
          float x, y;
          LOG("Initialising UAV.");

          // Initialise plane's vector
          plane->real_pos = start;
          x = (float)sin( DEG_TO_RAD * heading );           // Start flying straight &
          level.
          y = (float)cos( DEG_TO_RAD * heading );
          plane->normal = Vector3D( start.x, start.y, 0.0 );
          plane->normal.normalise();
          plane->step =   Vector3D( airspeed * x, airspeed * y, 0.0 );
          plane->air_speed = airspeed;
          plane->roll = 0.0f;
          plane->roll_target = 0.0f;
          plane->floor = floor;
          plane->ceiling = ceiling;
          plane->gps_delay = gps;
          plane->heading_target = heading;

          // Build search pattern:
          Pattern = Pattern_t ( MapSearchWidth, MapSearchEnabled);
          return Pattern.BuildSearchPattern( pmap );
}

void FlyStep( UAV_t *plane, Map_t * pmap, float deltaT)
{
          static float v_factor = 1, skip_time = 0, time_in_lift = 0;
          static int iCurrentTurnWeight = 0;
          float turn_ang, theta, alpha, in_lift, alpha_fix, heading_fix;
          SimData_t res;
          bool gps_success;
          Vector3D to_real_pos;


          /* Get VAV data and store into map. Overwrite any existing data. */
          if( GetSimData(plane, &res, deltaT, v_factor, in_lift) == 0 )
          {
              if( in_lift != 0 ) time_in_lift += deltaT;
```

```
    // Update the (real)position & roll of the UAV.
    plane->real_pos = res.position;
    plane->roll += res.Roll;

    // Record the position in the GPS list
    GPSElement_t *gple = new GPSElement_t( res.position, world_time + plane-
>gps_delay );
    gpl << gple;
    // Get the plane's position from GPS.
    gps_success = gpl.current( world_time, plane->gps_pos );  // if returns
false, we don't know where we are :(

    LOGFILE( world_time << ", " << plane->real_pos.x << ", " << plane-
>real_pos.y << ", " << plane->real_pos.z << ", " << res.gnd_speed << ", " <<
in_lift << ", " << time_in_lift);

    theta = plane->step.theta();
    alpha = plane->step.alpha();
    alpha_fix = -alpha;

    // Get a suggestion about best way to turn.
    // Can only provide advice if we know where we are.
    if( gps_success )
    {
        // Store any learnt information in the internal map, using the GPS
KNOWN POSITION.
        StoreInMap((const long)plane->gps_pos.x, (const long)plane->gps_pos.y,
res.Lift, pmap);

        if( Pattern.Searching() )
        {
            cout << "Following search pattern to point " <<
Pattern.lCurrentPoint+1 << " of " << Pattern.lNumPoints << "." << endl;
            // fly the search pattern
            if( FlyTo( plane->gps_pos, Pattern, turn_ang ) )
            {
                Pattern.lCurrentPoint++;
            }
            else
                plane->heading_target = turn_ang;
        }
        if( !Pattern.Searching() && skip_time > 1)
        {
            if( TurnAdvise( pmap, plane, turn_ang ) == 0 )
                plane->heading_target = turn_ang;
            skip_time = 0;
        }
    }

    // Slowly turn towards heading_target:
    // Calculate heading adjustment
    heading_fix = BETA_NOUGHT * deltaT;    //this is tightest possible turn.
    v_factor = (2 * PLANE_TURNING_CIRCLE / (float)PLANE_CRUISE_SPEED) * sin(
DEG_TO_RAD*deltaT*BETA_NOUGHT ) / deltaT;

    if( theta > 180 ) theta -= 360;
    if( plane->heading_target > 180 ) plane->heading_target -= 360;
    turn_ang = (round( plane->heading_target - theta + 180) % 360)-180;

    if( turn_ang > BETA_NOUGHT*0.3 || turn_ang < -180 )
    {
```

```
            // turn right
            heading_fix *= 1;
            iCurrentTurnWeight = 1;
        }
        else if ( turn_ang < -BETA_NOUGHT*0.3 )
        {
            // turn left
            heading_fix *= -1;
            iCurrentTurnWeight = -1;
        }
        else
        {
            heading_fix = 0;    // no adjustment
            v_factor = 1;
            iCurrentTurnWeight = 0;
        }
        plane->roll_target = iCurrentTurnWeight * BETA_NOUGHT;    // need to roll
    to turn

        // Slowly try to fix the roll
        float roll_fix = plane->roll_target - plane->roll;
        if( roll_fix > ROLL_ADJUST_PER_STEP )
            roll_fix = ROLL_ADJUST_PER_STEP;          // left wing higher
        else if( roll_fix < -ROLL_ADJUST_PER_STEP )
            roll_fix = -ROLL_ADJUST_PER_STEP;         // right wing higher

        // Check that the UAV is above the altitude floor.

        if( plane->gps_pos.z < plane->floor && alpha < PITCH_ADJUST_PER_STEP )
            alpha_fix = PITCH_ADJUST_PER_STEP;            // nose up to increase
    altitude
        else if( plane->gps_pos.z > plane->ceiling && alpha > -
    PITCH_ADJUST_PER_STEP)
            alpha_fix = -PITCH_ADJUST_PER_STEP;

        // Apply pitch limit
        if( alpha_fix > PITCH_ADJUST_PER_STEP )
            alpha_fix = PITCH_ADJUST_PER_STEP;
        else if( alpha_fix < -PITCH_ADJUST_PER_STEP )
            alpha_fix = -PITCH_ADJUST_PER_STEP;



        //  Adjust the roll.
        plane->roll = (int)(plane->roll + roll_fix*deltaT) % 360;
        // Turn the UAV_t
        plane->normal.setangles( (alpha + alpha_fix)*deltaT, theta +
    heading_fix*deltaT, 1);
        // Change the direction of travel.
        plane->step.setangles( (alpha + alpha_fix) * deltaT, theta +
    heading_fix*deltaT, plane->air_speed);

        // Show some info:
        cout << "deltaT: " << deltaT << " Heading: " << (int)theta << " Suggested
    Heading: " << plane->heading_target << " Roll: " << (int)res.Roll << " Map:
    ";
#ifdef _DEBUG
        cout << (int)(100*pmap->PercentFull()) << "% full";
#endif
        cout << endl;
```

```
            cout << "x: " << (int)(plane->real_pos.x) << " y: " << (int)(plane-
        >real_pos.y) << " z: " << (int)(plane->real_pos.z) << " Gnd speed: " <<
        (int)(res.gnd_speed/deltaT) << endl;
            cout << " ------------------------------------------------------------------
        ---" << endl;
            //if( !terrain.bounds_check(plane->real_pos.x, plane->real_pos.y ))
        exit(0);
        }
        skip_time += deltaT;
}
/*-------------------------------------------------------------------------*/
/*        Function: BuildSearchPattern
        Parameters: pmap, a pointer to the Map_t internal map of the UAV.
        Returns:    true, if errors occured
                    false, if successful
        Purpose:
        This function uses the width and location of the map to generate a
        series of points that the UAV can follow to partially fill its map.
*/
bool Pattern_t::BuildSearchPattern( Map_t *pmap )
{
        if( !enabled )
        {
            LOG("Not using search pattern.");
        }
        else
        {
            LOG("Building search pattern.");
            int iLoops = 0.5 * pmap->ncol * pmap->dist_unit / (float)lGridWidth;

            lNumPoints = (iLoops+1)*4;
            Points = (long *) new long[this->lNumPoints*2];
            if( Points == NULL )
            {
                cerr << "Error allocating memory for search pattern." << endl;
                return false;
            }

            long top   = pmap->h_lat  + pmap->nrow * pmap->dist_unit;
            long right = pmap->h_long + pmap->ncol * pmap->dist_unit;
            long c = 0;
            for( long i = 0; i <= iLoops; i++ )
            {
                // left x, bottom y
                Points[c]   = pmap->h_long + i * lGridWidth;
                Points[c+1] = pmap->h_lat  + i * lGridWidth;
                c+=2;

                // left x, top y
                Points[c]   = Points[c-2];
                Points[c+1] = top - i * lGridWidth;
                c+=2;

                // right x, top y
                Points[c]   = right - i * lGridWidth;
                Points[c+1] = Points[c-1];
                c+=2;

                // right x, bottom y.
                Points[c]   = Points[c-2];
                Points[c+1] = pmap->h_lat  + i * lGridWidth;
```

```
                c+=2;
            }
        }
        return true;
}
//--------------------------------------------------------------------------
//              Pattern_t Class function definitions.
//
// Are there points yet to be visited?
bool Pattern_t::Searching( void )
{
        return enabled && (lCurrentPoint < lNumPoints);
}
/*--------------------------------------------------------------------------*/
Pattern_t& Pattern_t::operator=( Pattern_t &rh)
{
        this->enabled = rh.enabled;
        this->lCurrentPoint = rh.lCurrentPoint;
        this->mlGridWidth = rh.GridWidth();
        this->lNumPoints = rh.lNumPoints;
        this->Points = rh.Points;
        rh.Points = NULL;
        return *this;
}
/*--------------------------------------------------------------------------*/
Pattern_t::~Pattern_t()
{
        if( Points ) delete [] Points;
}
//              END of Pattern_t function definitions.
/*--------------------------------------------------------------------------*/
```

Control.h

```
#ifndef _MAIN_H_
#define _MAIN_H_

void FlyStep( UAV_t *plane, Map_t * map, float deltaT);
bool InitPilot( UAV_t *plane, Map_t * map, Vector3D start, float heading, float
        airspeed, long floor, long ceiling, long gps, float MAPSearchWidth, bool
        MAPSearchEnabled );


// move these into graphics.cpp ?
// no point haaving a separate file for them.

#endif
```

GPSList_t.cpp

```
/*--------------------------------------------------------------------------
        The GPSList files create a lsit that contains information returned from
        the GPS unit. The results are only valid after a certain time.
--------------------------------------------------------------------------
File name:      GPSList_t.cpp
Author:         Ashley Goodwin
Created:     9 September 2005
--------------------------------------------------------------------------*/
#include "GPSList_t.h"
```

```
// List element constructor
GPSElement_t::GPSElement_t() : position(), valid_at(0), next(NULL)
{
}
GPSElement_t::GPSElement_t( Vector3D pos, float va): position(pos), valid_at(va),
          next(NULL)
{}

// List constructor.
GPSList_t::GPSList_t() : head(NULL), tail(NULL)
{
}

GPSList_t::~GPSList_t()
{
          GPSElement_t *tmp;
          while( head != NULL )
          {
              tmp = head->next;
              delete head;
              head = tmp;
          }
}
// This operator used to add an element to the end of the list.
GPSList_t& GPSList_t::operator<<( GPSElement_t *element)
{
          if( head == NULL )
          {
              head = element;
              tail = element;
          }
          else
          {
              tail->next = element;
              tail = element;
          }
          element->next = NULL;
          return *this;
}
// remove the first element in the list.
GPSElement_t* GPSList_t::remove_head( void )
{
          if( head != NULL )     // If list is not empty...
          {
              if( head == tail )     // If there is only one element
              {
                  return head;           // List is now empty
                  head = tail = NULL;
              }
              else                   // Else...
              {
                  GPSElement_t *tmp;     // Remove element.
                  tmp = head;
                  head = head->next;

                  return tmp;
              }
          }
          return NULL;
}
```

```
// Return by reference a vector using most recently valid info.
// If no valid information, return false, otherwise true.
bool GPSList_t::current( float time_now, Vector3D &pos )
{
          // throw away elements at the head of the list that have been superceded.
          while( head != NULL )
          {
              if( head->next == NULL )
                  break;  // Stop when the second element is NULL
              if( head->next->valid_at >= time_now )
                  break;  // Stop when second element is invalid.

              delete remove_head();
          }

          // If no elements left...
          if( head == NULL )
              return false;

          if( head->valid_at <= time_now )
          {
              pos = head->position;
              return true;
          }

          return false;
}
//                END of GPSList_t function definitions.
/*------------------------------------------------------------------------*/
//                GPSTrack_t function definitions
bool GPSTrack_t::offset(float time_now, Vector3D &v )
{
          GPSElement_t *t;
          // throw away elements at the head of the list that have expired.
          while( head != NULL )
          {
              if( head->valid_at + delay >= time_now )
                  break;

              t = remove_head();
              offset_to_real_pos -= t->position;
              delete t;
          }
          v = offset_to_real_pos;
          return true;
}
/*------------------------------------------------------------------------*/
GPSTrack_t& GPSTrack_t::operator<<( GPSElement_t *element )
{
          offset_to_real_pos += element->position;
          GPSList_t::operator <<( element );
          return *this;
}
/*------------------------------------------------------------------------*/
GPSTrack_t::GPSTrack_t() : GPSList_t(),
          offset_to_real_pos( Vector3D(0, 0, 0) ), delay(0)
{         }
//                END of GPSTrack_t function definitions
/*------------------------------------------------------------------------*/
```

GPSList_t.h

```
/*-------------------------------------------------------------------------------
          The GPSList files create a lsit that contains information returned from
          the GPS unit. The results are only valid after a certain time.
-------------------------------------------------------------------------------
File name:       GPSList_t.h
Author:          Ashley Goodwin
Created:     9 September 2005
-------------------------------------------------------------------------------*/

#ifndef _GPSLIST_T_H_
#define _GPSLIST_T_H_

#include "vector.h"


class GPSElement_t
{
public:
          float          valid_at;
          Vector3D       position;
          GPSElement_t   *next;

          GPSElement_t();
          GPSElement_t( Vector3D pos, float va);
};

class GPSList_t
{
protected:
          GPSElement_t   *head;
          GPSElement_t   *tail;

          GPSElement_t* remove_head( void );
public:
          GPSList_t& operator<<( GPSElement_t *element);    // Add an element to the end
          of the list.
          bool current( float time_now, Vector3D &v );  // Return a vector using most
          recently valid info.

          GPSList_t();
          ~GPSList_t();
};

class GPSTrack_t : private GPSList_t
{
private:
          Vector3D offset_to_real_pos;
public:
          float delay;

          bool offset(float time_now, Vector3D &v );
          GPSTrack_t& operator<<( GPSElement_t *element );
          GPSTrack_t();
};

#endif // _GPSLIST_T_H_
```

Graphics.cpp

```
/*-------------------------------------------------------------------------
          The RING files create and maintain a list of vertices to be drawn by
          OpenGL. It chooses the vertices using a viewing frustum.

          The terrain is divided into cells and associated vertices are calculated
          and stored.
---------------------------------------------------------------------------
File name:       ring.cpp
Author:          Ashley Goodwin
Created:     9 June 2005
---------------------------------------------------------------------------*/
#include <iostream>
#include <stdlib.h>
#include <GL/glut.h>
#include <math.h>

using namespace std;
#pragma warning (disable: 4786)


#include "trace.h"
#include "UAV.h"
#include "sim_map.h"

#define cosf(a)   (float)cos((float)a)
#define sinf(a)   sin((float)a)
#define sqrtf(a)     sqrt((float)a)
#define expf(a)   exp((float)a)

#define M_SQRT2          (1.41421356237309504880168872420969)

#define RING_SIZE   1000    /* Size of ring */

//-----------------------------------------------------------------------------------------
          ---
// All these #defs moved to sim_map.h
//#define TerrainWidth    (3000)
//#define CellDim         (4)         /* Terrain cell is CellDim X Celldim quads */
//#define NumCells        (20)        /* Terrain grid is NumCells X NumCells cells */
//#define CellWidth       (TerrainWidth/NumCells)
//-----------------------------------------------------------------------------------------
          ---

// Draw_object flags indicating what the draw_object represents.
#define DRAW_TERRAIN_CELL    1
#define DRAW_UAV_t           2

typedef struct RenderRing_t
{
    unsigned int head, tail;
    DrawObj_t    **ring;
} render_ring;

render_ring      ringbuffer;
DrawObj_t    *drawPlane; // Draw_obj that holds drawing instructions for the UAV_t
extern Terrain_t terrain;


bool SphereInFrustum( float x, float y, float z, float radius );
void ExtractFrustum(void);
```

```
void MakeNormal(float v0[3], float v1[3], float v2[3], float *dest);


/*-------------------------------------------------------------------------*/
/*        Function: AddObjToRing, GetObjFromRing
          Parameters: pobj, a pointer to a DrawObj_t struct
          Returns:    AddObjToRing-
                          bool indicating if item could be added.
                      GetObjFromRing-
                          a pointer to a prefobj_t struct from the ring, NULL if
                          empty.
          Purpose:
          Add and remove pointers to DrawObj_t structs from the draw ring.
*/
bool AddObjToRing(DrawObj_t* pobj)
{
          if( ringbuffer.head == RING_SIZE+ringbuffer.tail-1 ) {return false;}
          ringbuffer.ring[ringbuffer.head % RING_SIZE] = pobj;
          ringbuffer.head++;
          return true;
}
DrawObj_t* GetObjFromRing(void)
{
    DrawObj_t *pobj;

    if( ringbuffer.tail == ringbuffer.head )
          {
              return NULL;    //it's empty
          }
    pobj = ringbuffer.ring[ringbuffer.tail % RING_SIZE];
    ringbuffer.tail++;
    return pobj;
}
/*-------------------------------------------------------------------------*/
/*        Function: GetCell
          Parameters: x,y - the bootom-left corner of the cell
          Returns:    a pointre to a DrawObj_t
          Purpose:
          Dynamically allocates memory for the cell, determines the values of the
          vertices within the cell & determines the colours.
*/
DrawObj_t* GetCell(int x, int y)
{
          DrawObj_t *obj;
          float i,j;
          float *p, *q, a;
          float minz, maxz;

          // make the memory for a terrain cell
          // a cell is CellDIM x CellDim sectors/subcells
          obj = (DrawObj_t *)calloc( 1, sizeof(DrawObj_t) );
          if( obj == NULL )
          {
              std::cerr << "Can't malloc memory for terrain cell." << std::endl;
              exit(10);
          }
          // now get memory for cell's vertices etc.
          obj->flags = (unsigned int *)calloc( 1, sizeof(unsigned int) );
          if( obj->flags == NULL )
          {
```

```
    std::cerr << "Couldn't allocate memory for terrain cell flags." <<
std::endl;
    exit(10);
}
obj->flags[0] = DRAW_TERRAIN_CELL;
obj->vdata = (GLfloat *)calloc( (CellDim + 1) * (CellDim + 1) * 3,
sizeof(obj->vdata) );
if( obj->vdata == NULL )
{
    std::cerr << "Couldn't allocate memory for terrain vertices." <<
std::endl;
    exit(10);
}
obj->cdata = (GLfloat *)calloc( (CellDim + 1) * (CellDim + 1) * 3,
sizeof(obj->cdata) );
if( obj->cdata == NULL )
{
    std::cerr << "Couldn't allocate memory for vertices' colours." <<
std::endl;
    exit(10);
}
obj->ndata = (GLfloat *)calloc( CellDim * CellDim * 3, sizeof( obj->ndata )
);
if( obj->ndata == NULL )
{
    std::cerr << "Couldn't allocate memory for normal vectors." << std::endl;
    exit(10);
}
obj->tdata = NULL; // no texture data

// go over cell and get vertices & normals
p = obj->vdata;
q = obj->cdata;
minz = maxz = terrain.altitude(x, y);
float max = terrain.max_altitude();
for( i = x; i <= x + CellWidth; i += CellWidth/CellDim ) //from x to
x+cellwidth in celldim steps
{
    // Move up the y axis.
    for( j = y; j <= y + CellWidth; j+=CellWidth/CellDim )
    {
        a = terrain.altitude(i,j);
        if( a > maxz ) maxz = a;
        if( a < minz ) minz = a;

        *(p++) = i;
        *(p++) = j;
        *(p++) = a;

        a /= max;   //normalise to guide colouring

        // Choose colour based on altitude.
        if (a <.000001)
        {
       *(q++) = 0.2;        // greeny ground
       *(q++) = 0.2;
       *(q++) = 0.05;
        }
        else if (a <.90)
        {
       *(q++) = a * .5;     // green to red
```

```
                *(q++) = (1.0 - a) *.4 + .1;
                *(q++) = 0.1;
            }
                 else
                 {
                *(q++) = a-0.1;
                *(q++) = a-0.1;         // white snow
                *(q++) = a-0.1;
            }
              }
          }
          obj->midz = (maxz - minz) / 2.0;
          return obj;
}
/*---------------------------------------------------------------------------*/
/*        Function: InitCells
          Parameters: void
          Returns:    bool, indicating success or failure
          Purpose:
          CAlls a function to generate each cell, calculates the normals within
          cells, allocate memory for the UAV_t's DrawObj_t, and builds the UAV_t.
*/
bool InitCells( void )
{
          int x, y;
          // Create the cells of terrain
          terrain.cells = new DrawObj_t * [NumCells*NumCells];
          if( terrain.cells == NULL )
          {
              std::cerr << "Memory allocation error for terrain.cells array." <<
          std::endl;
              return false;
          }

          LOG("Creating terrain vertices.");
          for (x = 0; x < NumCells; x++)
              for (y = 0; y < NumCells; y++)
              {
                  terrain.cells[x * NumCells + y] = GetCell(x*CellWidth,y*CellWidth);
                  // each cell holds the terrian info of that section of the map
                  // each cell is dynamically created.

                  DrawObj_t *cell = terrain.cells[x * NumCells + y];
                  float *curr = cell->vdata;
                  float *next = cell->vdata + 3 * (CellDim + 1);
                  GLfloat *p = cell->ndata;

                  // Go up each column of subcells
                  for( int i = 0; i < CellDim; i++ )
                  {
                      for( int j = 0; j < CellDim ; j++ )
                      {
                          // specify normal vector:
                          MakeNormal( curr, next, curr+3, p );
                          p+=3;

                          curr+=3;
                          next+=3;
                      }
                      curr+=3;
                      next+=3;
```

```
        }
    }
    // finished creating cells


// Create the draw ring.
ringbuffer.ring = (DrawObj_t **)calloc(RING_SIZE, sizeof(DrawObj_t *));
ringbuffer.head = ringbuffer.tail = 0;

// Create the UAV_t draw token
drawPlane = (DrawObj_t *)malloc( sizeof( DrawObj_t ) );
if( drawPlane == NULL )
{
    std::cerr << "Memory allocation error for plane draw object." <<
std::endl;
    exit(10);
}
drawPlane->vdata = (float *)calloc( 3*(6*4 + 6) + 6 + 1, sizeof( float ) );
drawPlane->flags = (unsigned int *)calloc( 1, sizeof( unsigned int ) );
if( drawPlane->vdata == NULL || drawPlane->flags == NULL )
{
    std::cerr << "Memory allocation error for plane draw token." << std::endl;
    exit( 10 );
}
drawPlane->flags[0] = DRAW_UAV_t;
drawPlane->cdata = drawPlane->tdata = NULL;
// Build the plane.
drawPlane->midz = 0;
float *p = drawPlane->vdata + 7;
// main body piece
// back
*(p) = -0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH; *(p+2) = -
0.5 * PLANE_HEIGHT; p += 3;
*(p) =  0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH; *(p+2) = -
0.5 * PLANE_HEIGHT; p += 3;
*(p) =  0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH; *(p+2) =
0.5 * PLANE_HEIGHT; p += 3;
*(p) = -0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH; *(p+2) =
0.5 * PLANE_HEIGHT; p += 3;
// front
*(p) = -0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE; *(p+2) = -0.5 *
PLANE_HEIGHT; p += 3;
*(p) =  0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE; *(p+2) = -0.5 *
PLANE_HEIGHT; p += 3;
*(p) =  0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE; *(p+2) =  0.5 *
PLANE_HEIGHT; p += 3;
*(p) = -0.5 * PLANE_WIDTH; *(p+1) = PLANE_CENTRE; *(p+2) =  0.5 *
PLANE_HEIGHT; p += 3;
// left wing
// left tip
*(p) = -0.5 * PLANE_WINGSPAN; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5
* PLANE_WING_HEIGHT; p += 3;
*(p) = -0.5 * PLANE_WINGSPAN; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5
* PLANE_WING_HEIGHT; p += 3;
*(p) = -0.5 * PLANE_WINGSPAN; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5
* PLANE_WING_HEIGHT; p += 3;
*(p) = -0.5 * PLANE_WINGSPAN; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5
* PLANE_WING_HEIGHT; p += 3;
// body end
*(p) = -0.5 * PLANE_WIDTH; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5 *
PLANE_WING_HEIGHT; p += 3;
```

```
        *(p) = -0.5 * PLANE_WIDTH; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5 *
        PLANE_WING_HEIGHT; p += 3;
        *(p) = -0.5 * PLANE_WIDTH; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5 *
        PLANE_WING_HEIGHT; p += 3;
        *(p) = -0.5 * PLANE_WIDTH; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5 *
        PLANE_WING_HEIGHT; p += 3;
        // right wing
        // right tip
        *(p) = 0.5 * PLANE_WINGSPAN; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5
        * PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WINGSPAN; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5
        * PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WINGSPAN; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5
        * PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WINGSPAN; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5
        * PLANE_WING_HEIGHT; p += 3;
        // body end
        *(p) = 0.5 * PLANE_WIDTH; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5 *
        PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WIDTH; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) = -0.5 *
        PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WIDTH; *(p+1) =  0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5 *
        PLANE_WING_HEIGHT; p += 3;
        *(p) = 0.5 * PLANE_WIDTH; *(p+1) = -0.5 * PLANE_WING_WIDTH; *(p+2) =  0.5 *
        PLANE_WING_HEIGHT; p += 3;
        // rudder
        *(p) = -0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH;
        *(p+2) =  0.5 * PLANE_HEIGHT; p += 3;  // back bottom corners
        *(p) =  0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH;
        *(p+2) =  0.5 * PLANE_HEIGHT; p += 3;
        *(p) = -0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH +
        PLANE_RUDDER_LENGTH; *(p+2) =  0.5 * PLANE_HEIGHT; p += 3;// front bottom
        corners
        *(p) =  0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH +
        PLANE_RUDDER_LENGTH; *(p+2) =  0.5 * PLANE_HEIGHT; p += 3;
        *(p) = -0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH;
        *(p+2) =  0.5 * PLANE_HEIGHT + PLANE_RUDDER_HEIGHT; p += 3;//bcak top corners
        *(p) =  0.5 * PLANE_RUDDER_WIDTH; *(p+1) = PLANE_CENTRE - PLANE_LENGTH;
        *(p+2) =  0.5 * PLANE_HEIGHT + PLANE_RUDDER_HEIGHT; p += 3;

        return true;
}

/*------------------------------------ Cull -------------------------------*/
/*      Function: CullProc
        Parameters: plane, a pointer to the UAV_t
        Returns:   void
        Purpose:
        Evaluates for all objects ni the world whether they are visible at the
        current moment. If they are visible, their DrawObj_t is added to the
        ringbuffer, ready to be rendered.
*/
void CullProc(UAV_t *plane)
{
        int             x, y;
        //Begin cull to viewing frustrum
        ExtractFrustum();

        // Add visible cells to ring buffer
        for(x=0; x < NumCells ; x++)
            for(y=0; y < NumCells; y++ )
```

```
                if( SphereInFrustum( CellWidth*(x + 0.5), CellWidth*(y + 0.5),
          terrain.cells[x*NumCells + y]->midz, M_SQRT2 * CellWidth /*/ 1.5*/ ) )
                    AddObjToRing( terrain.cells[x*NumCells + y] );

          // Add plane to ring buffer
#define INDEX_X      1
#define INDEX_Y      2
#define INDEX_Z      3
#define INDEX_ALPHA  4
#define INDEX_THETA  5
#define INDEX_ROLL   6

          if( SphereInFrustum( plane->real_pos.x, plane->real_pos.y, plane->real_pos.z,
          PLANE_BOUNDING_RADIUS ) )
          {
              drawPlane->flags[0] = DRAW_UAV_t;
              drawPlane->vdata[INDEX_X] = plane->real_pos.x;
              drawPlane->vdata[INDEX_Y] = plane->real_pos.y;
              drawPlane->vdata[INDEX_Z] = plane->real_pos.z;
              drawPlane->vdata[INDEX_ALPHA] = plane->normal.alpha();
              drawPlane->vdata[INDEX_THETA] = plane->normal.theta();
              drawPlane->vdata[INDEX_ROLL] = plane->roll;
              AddObjToRing( drawPlane );
          }
}
/*------------------------------------ Draw -------------------------------*/
/*        Function: DrawProc
          Parameters: void
          Returns:    void
          Purpose:
          Takes each element from the ringbuffer and renders it.
*/
void DrawProc(void)
{
          void DrawObj(DrawObj_t *pobj);
          DrawObj_t      *to_draw;

          glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

          while (  (to_draw = GetObjFromRing()) != NULL  )
          {
              DrawObj(to_draw);
          }
}
/*-------------------------------------------------------------------------*/
/*        Function: DrawObj
          Parameters: pobj, pointer to a DrawObj_t to render.
          Returns:    void
          Purpose:
          Renders the object. Uses it's flag to determine what function to call
          to handle the rendering.
*/
void DrawObj(DrawObj_t *pobj)
{
          void DrawCell(DrawObj_t *in );
          void DrawPlane( DrawObj_t *in );

    unsigned int *flagsptr = (pobj->flags);

    switch (*flagsptr)
          {
```

```
        case DRAW_UAV_t:
            // draw plane
            DrawPlane( pobj );
            flagsptr += 1;
            break;
        case DRAW_TERRAIN_CELL:
            DrawCell( pobj );
            flagsptr += 1;
            break;

        default:
            std::cerr << "Bad flag in drawobj: " << (int)(*flagsptr) << std::endl;
            flagsptr++;
            break;
        }
}
/*-------------------------------------------------------------------------*/
/*      Function: DrawCell
        Parameters: in, a cell to be drawn
        Returns:    void
        Purpose:
        Uses the cell object's vdata to draw the vertices. Colournig uses
        the cdata field. Normals are supplied by ndata.
*/
void DrawCell(DrawObj_t *in )
{
        int i,j;
        float *curr = in->vdata;
        float *next = in->vdata + 3 * (CellDim + 1);
        float *clr = in->cdata;
        float *nextclr = in->cdata + 3 * (CellDim + 1);
        GLfloat *normal = in->ndata;

        // Go up each column of subcells
        for( i = 0; i < CellDim; i++ )
        {
            glBegin( GL_QUAD_STRIP );
                for( j = 0; j < CellDim + 1; j++ )
                {
                    // specify normal vector:
                    if( j != CellDim )
                    {
                        glNormal3fv( normal );
                        normal +=3 ;
                    }

                    glColor3fv( clr );
                    glVertex3fv( curr );

                    glColor3fv( nextclr );
                    glVertex3fv( next );

                    clr+=3;
                    nextclr+=3;
                    curr+=3;
                    next+=3;

                }
            glEnd();
        }
}
```

```
/*------------------------------------------------------------------------*/
/*        Function: DrawPlane
          Parameters: in, a plane DrawObj_t to be rendered
          Returns:    void
          Purpose:
          Renders the UAV_t by using the vdata field of 'in' to call 'box', etc.
          Colouring of the UAV_t is hardcoded here.
*/
void DrawPlane( DrawObj_t *in )
{
          void box( float *p);

          float *p = in->vdata + 7;
          glMatrixMode( GL_MODELVIEW );  // just to make sure :)
          glPushMatrix();

          // draw the UAV_t.
          glTranslatef( in->vdata[INDEX_X], in->vdata[INDEX_Y], in->vdata[INDEX_Z]);   //
          translate to position of UAV_t
          glRotatef( -in->vdata[INDEX_THETA], 0.0, 0.0, 1.0 );          //rotate around
          to heading, -ve angle clockwise
          glRotatef( -in->vdata[INDEX_ALPHA], 1.0, 0.0, 0.0 );          // rotate to get
          up angle, -ve angle points down
          glRotatef( in->vdata[INDEX_ROLL], 0.0, 1.0, 0.0 );            // rotate around
          y axis to get roll

          glBegin( GL_QUADS );
              glColor3f(0.0, 0.0, 1.0 );
              box(p);                       // body
              glColor3f(1.0, 1.0, 0.0 );
              box(p+3*8);                   // left wing
              box(p+2*3*8);                 // right wing
          glEnd();

          // rudder
          p += 3*3*8;
          glBegin( GL_TRIANGLES );
              glColor3f(1.0, 0.0,0.0);
              glNormal3f( 1.0, 0.0, 0.0);    // right side
              glVertex3fv( p+3 );
              glVertex3fv( p+9 );
              glColor3f(1.0, 0.0,1.0);
              glVertex3fv( p+15 );

              glColor3f(1.0, 0.0,0.0);           // left side
              glNormal3f( -1.0, 0.0, 0.0);
              glVertex3fv( p );
              glVertex3fv( p+6 );
              glColor3f(1.0, 0.0,1.0);
              glVertex3fv( p+12 );
          glEnd();
          glBegin( GL_QUADS );
              // back
              glColor3f(1.0, 0.0,0.0);
              glNormal3f( 0.0, -1.0, 0.0);
              glVertex3fv( p );
              glVertex3fv( p+3 );
              glColor3f(1.0, 0.0,1.0);
              glVertex3fv( p+15 );
              glVertex3fv( p+12 );
              // bottom
```

```
            glColor3f(1.0, 0.0,0.0);
            glNormal3f( 0.0, 0.0, -1.0);
            glVertex3fv( p+3 );
            glVertex3fv( p+9 );
            glVertex3fv( p+6 );
            glVertex3fv( p );
            // top
            glColor3f(1.0, 0.0,1.0);
            glNormal3f( 0.0, 1.0, 1.0);
            glVertex3fv( p+15 );
            glColor3f(1.0, 0.0,0.0);
            glVertex3fv( p+9 );
            glVertex3fv( p+6 );
            glColor3f(1.0, 0.0,1.0);
            glVertex3fv( p+12 );
        glEnd();
        glPopMatrix();

}

void box( float *p)
{
        // body - back
        glNormal3f( 0.0, -1.0, 0.0 );
        glVertex3fv( p );
        glVertex3fv( p+3 );
        glVertex3fv( p+6 );
        glVertex3fv( p+9 );
        // body - front
        glNormal3f( 0.0, 1.0, 0.0 );
        glVertex3fv( p+12 );
        glVertex3fv( p+15 );
        glVertex3fv( p+18 );
        glVertex3fv( p+21 );
        // body -v lt side
        glNormal3f( -1.0, 0.0, 0.0 );
        glVertex3fv( p+12 );
        glVertex3fv( p );
        glVertex3fv( p+9 );
        glVertex3fv( p+21 );
        // body - right side
        glNormal3f( 0.0, 1.0, 0.0 );
        glVertex3fv( p+3 );
        glVertex3fv( p+15 );
        glVertex3fv( p+18 );
        glVertex3fv( p+6 );
        // body - top
        glNormal3f( 0.0, 0.0, 1.0 );
        glVertex3fv( p+6 );
        glVertex3fv( p+18 );
        glVertex3fv( p+21 );
        glVertex3fv( p+9 );
        // body - bottom
        glNormal3f( 0.0, 0.0, -1.0 );
        glVertex3fv( p+3 );
        glVertex3fv( p+15 );
        glVertex3fv( p+12 );
        glVertex3fv( p );
}
/*-------------------------------------------------------------------------*/
/*        Function: MakeNormal
```

```
          Parameters: v0, v1, v2 - three vertices
                      normal, where to store the normal vector
          Returns:    void
          Purpose:
          Calculates an un-normalised normal vector from the passed vertices.
          The result is stored in normal.
*/
void MakeNormal(float v0[3], float v1[3], float v2[3], float *normal)
{
    GLfloat a[3], b[3];

    // note that v[3] is defined with counterclockwise winding in mind
    // a
    a[0] = v0[0] - v1[0];
    a[1] = v0[1] - v1[1];
    a[2] = v0[2] - v1[2];
    // b
    b[0] = v1[0] - v2[0];
    b[1] = v1[1] - v2[1];
    b[2] = v1[2] - v2[2];

    // calculate the cross product and place the resulting vector
    // into the address specified by *normal
    normal[0] = (a[1] * b[2]) - (a[2] * b[1]);
    normal[1] = (a[2] * b[0]) - (a[0] * b[2]);
    normal[2] = (a[0] * b[1]) - (a[1] * b[0]);

}

/*----------------------------------------------------------------------------*/
// These function are not my own.
/* Culling and Frustum testing routines by:
Mark Morley, December 2000
Originally posted on http://www.MarkMorley.com
which is no longer in existence.

Obtained via http://crownandcutlass.sourceforge.net/features/technicaldetails/
which sourced the page from Google's cache.
*/
float frustum[6][4]; //That's six sets of four numbers (six planes, each with an A, B,
          C, and D value).

void ExtractFrustum(void)
{
   float   proj[16];
   float   modl[16];
   float   clip[16];
   float   t;

   /* Get the current PROJECTION matrix from OpenGL */
   glGetFloatv( GL_PROJECTION_MATRIX, proj );

   /* Get the current MODELVIEW matrix from OpenGL */
   glGetFloatv( GL_MODELVIEW_MATRIX, modl );

   /* Combine the two matrices (multiply projection by modelview) */
   clip[ 0] = modl[ 0] * proj[ 0] + modl[ 1] * proj[ 4] + modl[ 2] * proj[ 8] + modl[
          3] * proj[12];
   clip[ 1] = modl[ 0] * proj[ 1] + modl[ 1] * proj[ 5] + modl[ 2] * proj[ 9] + modl[
          3] * proj[13];
```

```
clip[ 2] = modl[ 0] * proj[ 2] + modl[ 1] * proj[ 6] + modl[ 2] * proj[10] + modl[
       3] * proj[14];
clip[ 3] = modl[ 0] * proj[ 3] + modl[ 1] * proj[ 7] + modl[ 2] * proj[11] + modl[
       3] * proj[15];

clip[ 4] = modl[ 4] * proj[ 0] + modl[ 5] * proj[ 4] + modl[ 6] * proj[ 8] + modl[
       7] * proj[12];
clip[ 5] = modl[ 4] * proj[ 1] + modl[ 5] * proj[ 5] + modl[ 6] * proj[ 9] + modl[
       7] * proj[13];
clip[ 6] = modl[ 4] * proj[ 2] + modl[ 5] * proj[ 6] + modl[ 6] * proj[10] + modl[
       7] * proj[14];
clip[ 7] = modl[ 4] * proj[ 3] + modl[ 5] * proj[ 7] + modl[ 6] * proj[11] + modl[
       7] * proj[15];

clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[ 8] +
       modl[11] * proj[12];
clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[ 9] +
       modl[11] * proj[13];
clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj[10] +
       modl[11] * proj[14];
clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj[11] +
       modl[11] * proj[15];

clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[ 8] +
       modl[15] * proj[12];
clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[ 9] +
       modl[15] * proj[13];
clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj[10] +
       modl[15] * proj[14];
clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj[11] +
       modl[15] * proj[15];

/* Extract the numbers for the RIGHT plane */
frustum[0][0] = clip[ 3] - clip[ 0];
frustum[0][1] = clip[ 7] - clip[ 4];
frustum[0][2] = clip[11] - clip[ 8];
frustum[0][3] = clip[15] - clip[12];

/* Normalize the result */
t = sqrt( frustum[0][0] * frustum[0][0] + frustum[0][1] * frustum[0][1] +
       frustum[0][2] * frustum[0][2] );
frustum[0][0] /= t;
frustum[0][1] /= t;
frustum[0][2] /= t;
frustum[0][3] /= t;

/* Extract the numbers for the LEFT plane */
frustum[1][0] = clip[ 3] + clip[ 0];
frustum[1][1] = clip[ 7] + clip[ 4];
frustum[1][2] = clip[11] + clip[ 8];
frustum[1][3] = clip[15] + clip[12];

/* Normalize the result */
t = sqrt( frustum[1][0] * frustum[1][0] + frustum[1][1] * frustum[1][1] +
       frustum[1][2] * frustum[1][2] );
frustum[1][0] /= t;
frustum[1][1] /= t;
frustum[1][2] /= t;
frustum[1][3] /= t;

/* Extract the BOTTOM plane */
```

```
   frustum[2][0] = clip[ 3] + clip[ 1];
   frustum[2][1] = clip[ 7] + clip[ 5];
   frustum[2][2] = clip[11] + clip[ 9];
   frustum[2][3] = clip[15] + clip[13];

   /* Normalize the result */
   t = sqrt( frustum[2][0] * frustum[2][0] + frustum[2][1] * frustum[2][1] +
         frustum[2][2] * frustum[2][2] );
   frustum[2][0] /= t;
   frustum[2][1] /= t;
   frustum[2][2] /= t;
   frustum[2][3] /= t;

   /* Extract the TOP plane */
   frustum[3][0] = clip[ 3] - clip[ 1];
   frustum[3][1] = clip[ 7] - clip[ 5];
   frustum[3][2] = clip[11] - clip[ 9];
   frustum[3][3] = clip[15] - clip[13];

   /* Normalize the result */
   t = sqrt( frustum[3][0] * frustum[3][0] + frustum[3][1] * frustum[3][1] +
         frustum[3][2] * frustum[3][2] );
   frustum[3][0] /= t;
   frustum[3][1] /= t;
   frustum[3][2] /= t;
   frustum[3][3] /= t;

   /* Extract the FAR plane */
   frustum[4][0] = clip[ 3] - clip[ 2];
   frustum[4][1] = clip[ 7] - clip[ 6];
   frustum[4][2] = clip[11] - clip[10];
   frustum[4][3] = clip[15] - clip[14];

   /* Normalize the result */
   t = sqrt( frustum[4][0] * frustum[4][0] + frustum[4][1] * frustum[4][1] +
         frustum[4][2] * frustum[4][2] );
   frustum[4][0] /= t;
   frustum[4][1] /= t;
   frustum[4][2] /= t;
   frustum[4][3] /= t;

   /* Extract the NEAR plane */
   frustum[5][0] = clip[ 3] + clip[ 2];
   frustum[5][1] = clip[ 7] + clip[ 6];
   frustum[5][2] = clip[11] + clip[10];
   frustum[5][3] = clip[15] + clip[14];

   /* Normalize the result */
   t = sqrt( frustum[5][0] * frustum[5][0] + frustum[5][1] * frustum[5][1] +
         frustum[5][2] * frustum[5][2] );
   frustum[5][0] /= t;
   frustum[5][1] /= t;
   frustum[5][2] /= t;
   frustum[5][3] /= t;
}

bool PointInFrustum( float x, float y, float z )
{
   int p;

   for( p = 0; p < 6; p++ )
```

```
      if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <=
          0 )
         return false;
   return true;
}

bool SphereInFrustum( float x, float y, float z, float radius )
{
   int p;

   for( p = 0; p < 6; p++ )
      if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <=
          -radius )
         return false;
   return true;
}
/*---------------------------------------------------------------------------*/
```

Main.cpp

```
/*-----------------------------------------------------------------------------
          The Graphics file initialises and handles OpenGl graphics. Using a timer,
          it calls FlySteps to simulate aircraft movement. This file contains the
          main() function.
-----------------------------------------------------------------------------
File name:       graphics.cpp
Author:          Ashley Goodwin
Created:     21 April, 2005
-----------------------------------------------------------------------------*/
#include <string>
#include <iostream>
#include <math.h>
#include <GL/glut.h>

#include "trace.h"
#include "camera.h"
#include "UAV.h"
#include "sim_map.h"
#include "map.h"
#include "control.h"
#include "advice.h"

bool InitCells( void  );
void CullProc(UAV_t *);
void DrawProc(void);
void DrawCube( float vert[8][3] );

void grid( int height, int spacing, int width, int thickness );
void write_string( std::string );

// There is real time and simulation time.
long timer_period = 100; // Real time between stepping simulation time by dt.
float deltaT = 0.5f;  // Amount to advance simulation time by. Sim-time is advanced
                              //     by deltaT every timer_period.

float world_time = 0.0f;

// run time options
```

```
bool flagCameraFollow = true;
bool flagDrawLiftSegements = false;
bool flagPauseUAVMovement = false;
bool flagShowGrid = false;
bool flagShowSphere = false;
int  iLookMultiplier = 1;

extern Map_t  map;
UAV_t            plane;
Camera_t     objCamera;
extern           Terrain_t terrain;

/*--------------------------------------------------------------------------*/
/*        Function: TimerFunc
          Parameters: num - this value is the same as specified by the
                          glutTimerFunc callback registration call.
          Returns:    void
          Purpose:
          Moves the UAV_t a step in its current heading to simulate flight.
          TimerFunc is called roughly every 'timer_period' milliseconds.
*/
void TimerFunc( int num )
{
          // Has UAV crashed?
          if( plane.real_pos.z <= terrain.altitude( plane.real_pos.x, plane.real_pos.y
          ) )
          {
              // Yep :)
              cout << "The UAV has crashed." << endl;
              exit(0);
          }
          // Fly one step in current direction.
          if( !flagPauseUAVMovement )
          {
              TRACE("Stepping.");
              FlyStep( &plane, &map, deltaT);
          }

          world_time += deltaT;

          // Refresh the display.
          glutPostRedisplay();
          // Call this timer again.
          glutTimerFunc( timer_period, TimerFunc, num);
}
/*--------------------------------------------------------------------------*/
/*        Function: InitGraphics
          Parameters: void
          Returns:    void
          Purpose:
          Sets up parameters for the OpenGL graphics.
    Initialize antialiasing for RGBA mode, including alpha
          blending, hint, and line width.
*/
void InitGraphics(void)
{
          LOG("Initialising graphics.");
          // Set up GLUT & window.
          // Double buffered, RGB style colour.

          glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

```
        glutInitWindowSize(700, 550);
        glutInitWindowPosition(100, 0);
        glutCreateWindow("UAV Lift Seeker");

        glEnable(GL_DEPTH_TEST);                            // Enable depth buffering.
        glDepthFunc(GL_LESS);

        glShadeModel(GL_SMOOTH);                            // Enable Smooth Shading
        glClearDepth(1.0f);                                     // Depth Buffer Setup
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);    // Really Nice
        Perspective Calculations

        glClearColor(0.5, 0.5, 1.0, 1.0);                   // Washed out blue colour
        glEnable( GL_NORMALIZE);
        glLineWidth( 6.0f);

        // Specify the lighting.
        glEnable( GL_LIGHTING );

        GLfloat Light0Position[]= { 3000.0f, 3000.0f, 1000.0f, 1.0f };
        GLfloat ambient[] = { 0.5f, 0.5f, 0.5f, 1.0f };
        GLfloat diffuseLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

        glEnable(GL_LIGHT0);
        glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_POSITION, Light0Position);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);

        // Enable color tracking - glColor sets material colour
        glEnable(GL_COLOR_MATERIAL);
        glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

        // Initial camera position
                            //   Position       View(target)    Up
        objCamera.PositionCamera(  0, 0, 2500,    10, 0, 2500,    0.0, 0.0, 1.0);
}
/*-------------------------------------------------------------------------*/
/*      Function: Display
        Parameters: void
        Returns:    void
        Purpose:
        The Display function is called by OpenGL when screen needs redrawing.
    It draws the plane's position.
*/
void Display(void)
{
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        if( flagCameraFollow )
        {
            static Vector3D view = Vector3D( 0, iLookMultiplier, -0.6);
            // Follow at 90 degree blocks
            float theta = plane.step.theta();
            // pointing up/down
            if( (theta < 45 - 0.6*BETA_NOUGHT || theta > 315 + 0.6*BETA_NOUGHT) ||
        (theta > 135 + 0.6*BETA_NOUGHT && theta < 225 - 0.6*BETA_NOUGHT) )
            {
                view = Vector3D( 0, iLookMultiplier, -0.6);
            }
```

```
    // pointing right/left
    else if( (theta > 50 && theta < 130) || (theta > 230 && theta < 310) )
    {
        view = Vector3D( iLookMultiplier, 0, -0.6);
    }
    view = view.normal() * (objCamera.mPos - objCamera.mView).length();

    objCamera.mPos = plane.real_pos - view;
    objCamera.mView = plane.real_pos;
}

gluLookAt(objCamera.mPos.x,  objCamera.mPos.y,  objCamera.mPos.z,
          objCamera.mView.x, objCamera.mView.y, objCamera.mView.z,
          objCamera.mUp.x,   objCamera.mUp.y,   objCamera.mUp.z);

CullProc( &plane );    // Adds visible items to the draw ring.
DrawProc();        // Displays the draw ring.

// draw the lift segments
if( flagDrawLiftSegements )
{
    vector<LiftSegment_t>::iterator itr = terrain.m_lift.begin();
    while( itr != terrain.m_lift.end() )
    {
        float vert[8][3] ={    {itr->endpoint[0].x, itr->endpoint[0].y,itr-
>endpoint[0].z},
                               {itr->endpoint[0].x, itr->endpoint[0].y-10,itr-
>endpoint[0].z},
                               {itr->endpoint[1].x, itr->endpoint[1].y-10,itr-
>endpoint[1].z},
                               {itr->endpoint[1].x, itr->endpoint[1].y,itr-
>endpoint[1].z},
                               {itr->endpoint[0].x, itr->endpoint[0].y,itr-
>endpoint[0].z-10},
                               {itr->endpoint[0].x, itr->endpoint[0].y-10,itr-
>endpoint[0].z-10},
                               {itr->endpoint[1].x, itr->endpoint[1].y-10,itr-
>endpoint[1].z-10},
                               {itr->endpoint[1].x, itr->endpoint[1].y,itr-
>endpoint[1].z-10}     };
        // draw the segment
        if( itr->type == LiftSegment_t::RISE )
            glColor3f( 1.0, 0.0, 0.0);
        else
            glColor3f( 0.0, 0.0, 1.0 );
        DrawCube( vert );
        itr++;
    }
}
if( flagShowGrid ) grid( 1100, 100, terrain.width, 5);
if( flagShowSphere )
{
    if( map.BoundsCheck( plane.real_pos ) )
        glColor3f( 1, 0, 0);
    else
        glColor3f( 0, 1, 0);
    glPushMatrix();
    glTranslatef( plane.real_pos.x, plane.real_pos.y, plane.real_pos.z );
    glutSolidSphere( 3, 20, 20 );
    glPopMatrix();
}
```

```
        glutSwapBuffers();         // Display the current (modified) buffer.
}
/*--------------------------------------------------------------------------*/
/*      Function: Reshape
        Parameters: w - the new window width.
                    h - the new window height.
        Returns:    void
        Purpose:
        The Reshape function is called by OpenGL to handle window resizes.
        It initialises the coordinate space.
*/
void Reshape(int width, int height)
{
        LOG("Executing window reshape function.");
        // Set veiewing area to whole window.
        glViewport(0, 0, width, height);
        // Set up standard eye angle & position.
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(90.0, (float)width / height, 1, 10000);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        // Clear the display.
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glutPostRedisplay();
}
/*--------------------------------------------------------------------------*/
/*      Function: Keyboard
        Parameters: key - the generated ASCII character.
                    x, y - the position of the mouse pointer when the key was
                        pressed.
        Returns:    void
        Purpose:
        The Keyboard function is called to handle keypresses.
*/

void Keyboard(unsigned char key, int x, int y)
{
        switch (key)
        {
        case ' ':
            iLookMultiplier *= -1;
            break;
        case 's':
        case 'S':   // toggle sphere
            flagShowSphere = !flagShowSphere;
            break;
        case 'g':   // toggle the grid
        case 'G':
            flagShowGrid = !flagShowGrid;
            break;
        case 'p':   // toggle UAV_t movement on/off
        case 'P':
            flagPauseUAVMovement = !flagPauseUAVMovement;
            break;
        case 'l':   // toggle lift segment rendering
        case 'L':
            flagDrawLiftSegements = !flagDrawLiftSegements;
```

```
            break;
          case 'f':   // toggle camera following
          case 'F':
              flagCameraFollow = !flagCameraFollow;
              break;

          case 'q':   // quit the program
    case 'Q':
    case 27: //  Escape Key
        exit(0);
        break;

          case '+':   // increase the speed of the simulation
          case '=':   // i.e. do more steps, of the same deltaT
              timer_period -= 50;
              if( timer_period < 10 ) timer_period = 10;
              break;
          case '-':   // decrease the speed of the simulation
                      // i.e. do less steps, of the same deltaT
              timer_period += 50;
              break;

          case ']':   // increase the size of deltaT
              deltaT += 0.1f;
              if( deltaT > 1.0f ) deltaT = 1.0f;
              break;
          case '[':   // decrease the size of deltaT
              deltaT -= 0.1f;
              if( deltaT < 0.1f) deltaT = 0.1f;
              break;

          default:    // ignore unhandled keys
        break;
    }
          glutPostRedisplay();
}
void Keyboard2(int key, int x, int y)
{
          switch(key)
          {
          case GLUT_KEY_UP:      // move forward
              objCamera.MoveCamera( 10*CAMERASPEED);
              break;
          case GLUT_KEY_DOWN:        // move backward
              objCamera.MoveCamera(-10*CAMERASPEED);
              break;
          case GLUT_KEY_LEFT:        // turn left
              objCamera.RotateView( CAMERASPEED);
              break;
          case GLUT_KEY_RIGHT:   // turn right
              objCamera.RotateView(-CAMERASPEED);
              break;
          case GLUT_KEY_PAGE_UP: // look up
              objCamera.AngleView(8);
              break;
          case GLUT_KEY_PAGE_DOWN:// look down
              objCamera.AngleView(-8);
              break;
          case GLUT_KEY_HOME:        // move upwards
              objCamera.mPos.z += 10;
              objCamera.mView.z += 10;
```

```
            break;
        case GLUT_KEY_END:     // move downwards
            objCamera.mPos.z -= 10;
            objCamera.mView.z -= 10;
            break;

        default:    // ignore unhandled keys
            break;
        }
        glutPostRedisplay();
}
/*-------------------------------------------------------------------------*/
/*      Function: main
        Parameters: argc - the number of command line arguments.
                    argv - an array of the arguments as character strings.
        Returns:    int
        Purpose:
        This function is the entry point for execution. Command line arguments
        are processed to prepare the simulation.

        It calls initialisation functions for the UAV_t and OpenGL. Control is
        passed to OpenGL after all initialisation is complete. The program
        relies upon a timer callback to continue execution.
*/
int main(int argc, char** argv)
{
        using namespace std;
        float head = 0;
        float wa = 0;   // defalut wind values - probably not needed.
        float ws = 7;
        float wf = 0.0;

        float WINDSpeed = 7, WINDAngle = 0, WINDFluctuation = 0, UAVStartHeading = 0;
        long  UAVFloor = 0, UAVStartX = 0, UAVStartY = 0, UAVStartZ = 600,
        UAVGPSDelay = 3,
                MAPWidth = 50, MAPHeight = 50, MAPSeparation = 20, MAPSearchWidth =
        300,
                MAPLongitude = 10, MAPLatitude = 10, UAVCeiling = 10000,
        MAPEscapeDistance = 0;
        bool  MAPSearchEnabled = true;

        Tracing( false );
        Logging( true  );

        // Initialise the OpenGL graphics library.
        glutInit(&argc, argv);
        // Initialise the graphics.
        InitGraphics();
        // Read in the simulation settings.
        string strfile("data\\map.txt");
        if( !FetchTerrain( strfile, WINDSpeed, WINDAngle, WINDFluctuation,
                UAVFloor, UAVCeiling, UAVStartHeading, UAVStartX, UAVStartY,
        UAVStartZ, UAVGPSDelay, MAPWidth,
                MAPHeight, MAPLongitude, MAPLatitude, MAPSeparation, MAPSearchWidth,
        MAPSearchEnabled, MAPEscapeDistance )    )
            return 1;
        // Initialise the UAV.
        Vector3D UAVStart(UAVStartX, UAVStartY, UAVStartZ);
        // do the map...
        map = Map_t( MAPWidth, MAPHeight, MAPLongitude, MAPLatitude, MAPSeparation,
        MAPEscapeDistance );
```

```
        if( InitPilot(&plane, &map, UAVStart, UAVStartHeading, PLANE_CRUISE_SPEED,
        UAVFloor, UAVCeiling, UAVGPSDelay, MAPSearchWidth, MAPSearchEnabled) == false
        )
            return 2;

        if( InitCells() == false )
            return 3;

        SetWindParam( WINDSpeed, WINDAngle, WINDFluctuation);

        LOGFILE("MAP long, " << map.h_long);
        LOGFILE("MAP lat, " << map.h_lat);
        LOGFILE("Map width, " << MAPSeparation * MAPWidth);
        LOGFILE("Map height, " << MAPSeparation * MAPHeight);
        LOGFILE("Pattern width: " << MAPSearchWidth);

        LOGFILE("Advice length: " << Circle_SearchDist);
        LOGFILE("Advice # angles: " << Circle_NumAngles);


        LOGFILE("time, x, y, z, gnd speed, lift amount, time in lift");

        glutReshapeFunc(Reshape);      //TODO: check error value of these fns.
        glutKeyboardFunc(Keyboard);
        glutSpecialFunc(Keyboard2);

        glutDisplayFunc(Display);

        // Initialisation complete. Start doing the good stuff!
        LOG("Initialisation complete.");
        LOG("-----------------------------------------------------------------------");

        // glutTimerFunc( triggering time in ms, callback func with one int argument,
        integer id value which is passed to callback_fn );
        glutTimerFunc( timer_period, TimerFunc, 1);
        glutMainLoop();
   return 0;
}
// Draws a cube using current matrices & colour. The 8 vertices
// are passed in an array.
/*
          3 .--------. 2
              .        . |
            0 .--------.   |
              |        |   |
              |        |   . 6
              |        | .
              .--------.
             4         5
          The vertices are ordered counter-clockwise as shown,
          starting with zero.
          (Imagine 1 is labeled - the pic looks silly with it included)
*/
void DrawCube( float vert[8][3] )
{
          //draw sixe faces:
          glBegin(GL_QUAD_STRIP);
              glVertex3f( vert[0][0], vert[0][1], vert[0][2]);  //first face
              glVertex3f( vert[4][0], vert[4][1], vert[4][2]);
              glVertex3f( vert[1][0], vert[1][1], vert[1][2]);
              glVertex3f( vert[5][0], vert[5][1], vert[5][2]);
```

```
            //other faces need only two points.
            glVertex3f( vert[2][0], vert[2][1], vert[2][2]);
            glVertex3f( vert[6][0], vert[6][1], vert[6][2]);

            glVertex3f( vert[3][0], vert[3][1], vert[3][2]);
            glVertex3f( vert[7][0], vert[7][1], vert[7][2]);

            glVertex3f( vert[0][0], vert[0][1], vert[0][2]);
            glVertex3f( vert[4][0], vert[4][1], vert[4][2]);
        glEnd();
        glBegin(GL_QUADS);
            // top and bottom faces
            glVertex3f( vert[0][0], vert[0][1], vert[0][2]);
            glVertex3f( vert[1][0], vert[1][1], vert[1][2]);
            glVertex3f( vert[2][0], vert[2][1], vert[2][2]);
            glVertex3f( vert[3][0], vert[3][1], vert[3][2]);

            glVertex3f( vert[4][0], vert[4][1], vert[4][2]);
            glVertex3f( vert[5][0], vert[5][1], vert[5][2]);
            glVertex3f( vert[6][0], vert[6][1], vert[6][2]);
            glVertex3f( vert[7][0], vert[7][1], vert[7][2]);
        glEnd();
}

void grid( int height, int spacing, int width, int thickness )
{
        glColor3f( 1, 0, 0);
        long x = 0, y = 0;
        for( ; x <= width; x += spacing )
        {
            float vert[8][3] = {   {x, 0, height},
                                   {x+thickness, 0, height},
                                   {x+thickness, width, height},
                                   {x, width, height},
                                   {x, 0, height+thickness},
                                   {x+thickness, 0, height+thickness},
                                   {x+thickness, width, height+thickness},
                                   {x, width, height+thickness}
                                  };
            DrawCube( vert );
        }
        for( ; y < width; y += spacing )
        {
                float vert[8][3] = {{0, y, height},
                                   {width, y, height},
                                   {width, y+thickness, height},
                                   {0, y+thickness, height},
                                   {0, y, height+thickness},
                                   {width, y, height+thickness},
                                   {width, y+thickness, height+thickness},
                                   {0, y+thickness, height+thickness}
                                  };
            DrawCube( vert );
        }
}
```

Map.cpp

```
/*-----------------------------------------------------------------------------
        The MAP files are a acollection of tools that can be used to manipulate
```

```
          the map of Vertical Air Velocities maintained in the system.
-------------------------------------------------------------------------------
File name:       map.cpp
Author:          Ashley Goodwin
Created:     9 March 2005
-------------------------------------------------------------------------------*/
#include <stdlib.h>

#include "trace.h"
#include "map.h"
/*-------------------------------------------------------------------------------*/
//               Map_t function definitions
Map_t::Map_t(unsigned int num_rows, unsigned int num_cols, long longitude, long
          latitude, int separation, long esc_dist)
: mdist_unit(separation), h_long(longitude), h_lat(latitude), ncol(num_cols),
          nrow(num_rows), escape_distance(esc_dist)
{
          /*  Function: Map_t::Map_t(...)   -  constructor
              Parameters: num_rows, the number of rows the map is to have.
                          num_cols, the number of columns the map is to have.
                          separation, the distance between the elements.
              Returns:    -nothing-
              Purpose:
              Allocates memory to the Map_Data array, as specified by the arguments.
              Rows and collumns are seperated by a distance of:
                  Map_t::dist_unit
          */
          unsigned int i;
          Map_t::Map_Data *pTemp = NULL;
          LOG("Constructing the map.");

          // Attempt to allocate memory for row pointers:
          if( (this->rows = (Map_t::Map_Data **)calloc(sizeof( Map_t::Map_Data* ),
          nrow)) == NULL )
          {
              // Error allocating memory.
              // Nothing to clean up.
              exit(1);
          }
          // Attempt to allocate memory to each of the rows.
          for(i = 0; i < nrow; i++)
          {
              if( (pTemp = (Map_t::Map_Data *)calloc(sizeof(Map_t::Map_Data),  ncol)) ==
          NULL)
              {
                  // Error allcoating memory. free already allocated mem and exit
          function.
                  i--;
                  while( i >= 0 )
                  {
                      free(this->rows[i]);   // Free all rows back to the start.
                      i--;
                  }
                  exit(2);                   // Return error.
              }
              this->rows[i] = pTemp;         // All is good, store row.
          }
#ifdef _DEBUG
          num_full = 0;
#endif
}
```

```
/*---------------------------------------------------------------------------*/
Map_t::~Map_t()  //destructor
{
          TRACE("Destructig the map.");
          if( rows == NULL ) return;
          unsigned long i;
          for(i = 0; i < Map_t::nrow; i++)
          {
              free( rows[i] );
          }
          free( rows );
}
/*---------------------------------------------------------------------------*/
// Copy constructor:
// take values to construct with from rh
Map_t::Map_t( const Map_t &rh )
: mdist_unit( ((class Map_t &)rh).DistUnit())
{
          unsigned int i;
          Map_t::Map_Data *pTemp = NULL;
          TRACE("Copying constructing the map.");

          // Set the number of rows and cols, etc.:
          this->ncol = rh.ncol;
          this->nrow = rh.nrow;
          this->h_lat = rh.h_lat;
          this->h_long = rh.h_long;
          this->escape_distance = rh.escape_distance;

          // Attempt to allocate memory for row pointers:
          if( (this->rows = (Map_t::Map_Data **)calloc(sizeof( Map_t::Map_Data* ), nrow
          )) == NULL )
          {
              // Error allocating memory.
              // Nothing to clean up.
              exit(1);
          }
          // Attempt to allocate memory to each of the rows.
          for(i = 0; i < nrow; i++)
          {
              if( (pTemp = (Map_t::Map_Data *)calloc(sizeof(Map_t::Map_Data),  ncol)) ==
          NULL)
              {
                  // Error allcoating memory. free already allocated mem and exit
          function.
                  i--;
                  while( i >= 0 )
                  {
                      free(this->rows[i]);   // Free all rows back to the start.
                      i--;
                  }
                  exit(2);                   // Return error.
              }
              this->rows[i] = pTemp;         // All is good, store row.
          }
#ifdef _DEBUG
          num_full = rh.num_full;
#endif
}
/*---------------------------------------------------------------------------*/
Map_t& Map_t::operator=( Map_t &rh )
```

```
{
          TRACE("Copying the map.");
          this->ncol = rh.ncol;
          this->nrow = rh.nrow;
          this->h_lat = rh.h_lat;
          this->h_long = rh.h_long;
          this->mdist_unit = ((class Map_t &)rh).dist_unit;
          this->rows = rh.rows;
          this->escape_distance = rh.escape_distance;
          rh.rows = NULL;
#ifdef _DEBUG
          num_full = rh.num_full;
#endif
          return *this;
}
/*-------------------------------------------------------------------------*/
bool Map_t::BoundsCheck( Vector3D point )
{
          return (    point.x > h_long && point.x < h_long + ncol * mdist_unit &&
                      point.y > h_lat  && point.y < h_lat  + nrow * mdist_unit );
}
//            END Map_t fucntion definitions
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/*        Function: WorldToMap, MapToWorld
          Parameters: p_long,    the longititude of the UAV.
                      p_lat,  the lattitude of the UAV.
                      col,    the column index (if valid) of the plane in the map.
                      row,    the row index (if valid) of the plane in the map.
                      map,    a pointer to the map of VAVs.
          Returns:    integer
                      0, if successfull.
                      >0, if errors occured.
          Purpose:
          WorldToMap takes a (longitude,lattitude) coordinate and determines if
          the coordinate is located within the map. If it is, the map indexes
          nearest to the coordinate are returned by reference, with the function
          returning 0 to indicate success. If the coordinate is not within the map,
          the function returns >0 to indicate failure.

          MapToWorld takes a map index (row,column) pair and converts it to a
          (longitude,lattitude) pair. Since this conversion cannot fail, it always
          returns 0, to indicate success.
*/
int WorldToMap( const long p_long, const long p_lat, int *col, int *row, Map_t * const
          map)
{
          long lat_diff  = (p_lat - map->h_lat);
          long long_diff = (p_long - map->h_long);

          // Use const int dist_unit to determine appropriate map coord/index.
          // Round the numbers to obtain closest index.
          long col_index = round((float)long_diff / map->dist_unit);
          long row_index = round((float)lat_diff  / map->dist_unit);

          // Check that both indices are in the valid range of the map. I.e.
          //  ([0,ROW_MAX],[0,COL_MAX])
          if( row_index >= 0 && col_index >= 0 && row_index < (signed)map->nrow &&
          col_index < (signed)map->ncol )
          {
              // Valid
```

```
            *col = col_index;
            *row = row_index;
            return 0;
        }
        // Else, return error.
        return 1;
}
// return the Map_Data element at the coords, if available.
Map_t::Map_Data* WorldToMap( const long p_long, const long p_lat, Map_t * const map)
{
        long lat_diff  = (p_lat - map->h_lat);
        long long_diff = (p_long - map->h_long);

        if( lat_diff < 0 || long_diff < 0 ) return NULL;

        // Use const int dist_unit to determine appropriate map coord/index.
        // Round the numbers to obtain closest index.
        long col_index = round(long_diff / (float)map->DistUnit());
        long row_index = round(lat_diff  / (float)map->DistUnit());

        // Check that both indices are in the valid range of the map. I.e.
        //  (  [0,ROW_MAX),  [0,COL_MAX)  )
        if( row_index < (signed)map->nrow && col_index < (signed)map->ncol )
        {
            // Valid    */
            return &(map->rows[row_index][col_index]);
        }
        // Else, return error.
        return NULL;
}
int MapToWorld( const int col, const int row, long *p_long, long *p_lat, Map_t * const
        map)
{       // Do the conversion directly.
        *p_long = (col * map->dist_unit) + map->h_long;
        *p_lat  = (row * map->dist_unit) + map->h_lat;

        return 0;
}
/*-------------------------------------------------------------------------*/
/*      Function: StoreInMap
        Parameters: p_long, the longitutde of the UAV.
                    p_lat, the lattitude of the UAV.
                    lift, the amount of lift as returned by the GPS.
                    map, a pointer to the map in which to store.
        Returns:    integer
                    0, if successfull.
                    >0, if errors occured.
        Purpose:
        To store data into the map, if the UAV is positioned within the map.
        Uses WorldToMap to determine map indices.
*/
int StoreInMap(const long p_long, const long p_lat, const long lift, Map_t * map)
{
        Map_t::Map_Data *p;
        // Check if location is valid
        if( (p = WorldToMap(p_long, p_lat, map)) != NULL )
        {
#ifdef _DEBUG
            if( !p->valid ) map->AddOne();
#endif
            // If so, store data
```

```
            p->lift = lift;
            p->valid = true;
            return 0;
        } // else error
        return 1;
}
```

Map.h

```
/*-------------------------------------------------------------------------
        The MAP files are a collection of tools that can be used to manipulate
        the map of Vertical Air Velocities maintained in the system.
-------------------------------------------------------------------------
File name:      map.cpp
Author:         Ashley Goodwin
Created:    9 March 2005
-------------------------------------------------------------------------*/
#ifndef _MAP_H_
#define _MAP_H_
#include "vector.h"

// returns nearest integer
// Portable?
inline int round(float in)
{
        return (int)(in+.5);
}

/*-------------------------------------------------------------------------*/
/*      Struct: Map
        Purpose:
        Holds information regarding the map's physical location and orientation.
        Also holds pointers to the map data.
*/
class Map_t
{
public:
        /*-------------------------------------------------------------------------
        */
        /*  Struct: Map_Data
            Purpose:
            Holds information at a location in the map.
        */
        class Map_Data
        {
        public:
            long lift;
            bool valid;
        };

        long    h_long, h_lat;  // The longitude and latitude of the map's home
        coordinate.
                                //   The home coordinate is index [0][0].
                                //   Longitude and lattitude are measured in metres.
                                // Longitude is east-west.
                                // Latitude is north-south.
        unsigned int ncol, nrow;// The number of rows and columns in the map.
        Map_Data  **rows;       // Holds pointers to the rows in the map. Each pointer
                                //          points to a row of Map_Data elements.
```

```
                                // The rows are dynamically allocated to allow less
        expensive
                                //          row destruction/creation/shifting.
        long    escape_distance;
private:
        int mdist_unit;         // The distance between each adjacent map element,
                                //      measured in metres.
#ifdef _DEBUG
private:
        int num_full;
public:
        void AddOne( void ) { num_full++; }
        float PercentFull( void ) { return num_full / (float)(nrow*ncol); }
#endif

public:
#define dist_unit DistUnit()

        int DistUnit(void) { return mdist_unit; } // Provides psuedo-protection for
        dist_unit
        bool BoundsCheck( Vector3D point );

        Map_t( const Map_t &rh);  // Copy constructor
        Map_t& operator=(  Map_t &);   // copy

        // Con/Destructor:
        Map_t(unsigned int num_rows, unsigned int num_cols, long longitude, long
        latitude, int separation, long esc_dist);
        ~Map_t();
} ;
/*-------------------------------------------------------------------------*/
/*      Function: WorldToMap, MapToWorld
        Parameters: p_long,    the longititude of the UAV.
                    p_lat,  the lattitude of the UAV.
                    col,    the column index (if valid) of the plane in the map.
                    row,    the row index (if valid) of the plane in the map.
                    map,    a pointer to the map of VAVs.
        Returns:    integer:
                    0, if successfull.
                    >0, if errors occured.

                    Map_t::MapData*:
                    If successful, pointer to the correspnding element.
                    Otherwise, NULL.

        Purpose:
        WorldToMap takes a (longitude,lattitude) coordinate and determines if
        the coordinate is located within the map. If it is, the map indexes
        nearest to the coordinate are returned by reference, with the function
        returning 0 to indicate success. If the coordinate is not within the map,
        the function returns >0 to indicate failure.

        MapToWorld takes a map index (row,column) pair and converts it to a
        (longitude,lattitude) pair. Since this conversion cannot fail, it always
        returns 0, to indicate success.
*/
int WorldToMap( const long p_long, const long p_lat, int *col, int *row, Map_t * const
        );
Map_t::Map_Data* WorldToMap( const long p_long, const long p_lat, Map_t * const );
            //return the Map_Data element at the coords, if available
```

```
int MapToWorld( const int col, const int row, long *p_long, long *p_lat, Map_t const
        *);
/*-------------------------------------------------------------------------*/
/*      Function: StoreInMap
        Parameters: p_long, the longitutde of the UAV.
                    p_lat, the lattitude of the UAV.
                    lift, the amount of lift as returned by the GPS.
                    map, a pointer to the map in which to store.
        Returns:    integer
                    0, if successfull.
                    >0, if errors occured.
        Purpose:
        To store data into the map, if the UAV is positioned within the map.
        Uses WorldToMap to determine map indexes.
*/
int StoreInMap(const long p_long, const long p_lat, const long lift, Map_t *);

#endif
```

Path.cpp

```
/*-------------------------------------------------------------------------
        The Path source files are used to search the map of VAVs and recommend
        a direction of travel in order to maximise time spent aloft.
---------------------------------------------------------------------------
File name:      path.cpp
Author:         Ashley Goodwin
Created:     10 April 2005
---------------------------------------------------------------------------*/
#pragma warning (disable: 4786)
#include <map>
#include <math.h>


using namespace std;

#include "trace.h"
#include "UAV.h"
#include "map.h"
#include "sim_map.h"
#include "path.h"
#include "advice.h"  // This header contains the configuration parameters.

float CircleSearch( Map_t *pmap, UAV_t *plane );
/*-------------------------------------------------------------------------*/
/*      Function: TurnAdvise
        Parameters: pmap - a pointer to the map to search.
                    plane - a pointer to the UAV_t structure.
                    ang - reference to result of function:
                            suggested heading.
                    fCurrentTurnWeight - positive if turning tu right, negative
                            if turning left. Lift in direction of current turn
                            is given extra weighting.
        Returns:    integer
                    0, if successful.
                    >0, if errors.
        Purpose:
        Use current data and map data to decide which direction to turn to
        acheive maximal lift. Returns by reference the suggested heading.
        The suggested heading is positive.
```

```
*/
int TurnAdvise( Map_t * pmap, UAV_t *plane, float &ang )
{
          float circle_advice_angle  = 0.0f;
          ang = (float)plane->step.theta();
          // Check wheter the UAV is way away from the map
          if( pmap->escape_distance > 0 &&
              (   (plane->gps_pos.x - pmap->h_long - 0.5*pmap->ncol*pmap-
          >DistUnit())*(plane->gps_pos.x - pmap->h_long - 0.5*pmap->ncol*pmap-
          >DistUnit())
                  + (plane->gps_pos.y - pmap->h_lat - 0.5*pmap->nrow*pmap-
          >DistUnit())*(plane->gps_pos.y - pmap->h_lat - 0.5*pmap->nrow*pmap-
          >DistUnit())
                  > pmap->escape_distance
              )
            )
          {   // It is, so head back toward the centre of the map.
              ang = 90 - atan2( (pmap->h_lat + 0.5*pmap->nrow*pmap->DistUnit()- plane-
          >gps_pos.y), (pmap->h_long + 0.5*pmap->ncol*pmap->DistUnit() - plane-
          >gps_pos.x) ) * RAD_TO_DEG;
          }
          else
          {
              // otherwise...
              circle_advice_angle = CircleSearch( pmap, plane);

              // Combine the advice received to choose angle of advice.
              if( circle_advice_angle > 0.0f )
              {
                  ang = (int)(ang+circle_advice_angle)%360;
                  LOG("Circle advice says: " << (int)ang%360 );
              }
          }
          // make sure ang is positive.
          if( ang < 0 ) ang += 360.0f;

          return 0;
}

// CircleTest is called by CircleSearch
float CircleTest( Map_t *pmap, UAV_t *plane, float fSearchAngle, Vector3D vStartOffset)
{         // Passed angle is heading offset.
          // This function performs a straight line test from vStartPor in the
          direction
          //  of (plane.heading + fSearchAngle). It searches for a distance of
          //  Next_SearchDist.

          Vector3D vStep;
          int     iStepNum, iNumSteps;
          Map_t::Map_Data *element;
          float lift;
          // Move the test point along the path.
          // Accumulate lift as the test point moves.
          // Move in steps of map->dist_unit. There is no point stepping smaller.
          fSearchAngle = 90 - fSearchAngle;
          vStep.setangles( 0.0f, plane->step.theta() + fSearchAngle, pmap->dist_unit );
          iNumSteps = Circle_SearchDist / (float)pmap->dist_unit;
          lift = 0.0f;
          vStartOffset += plane->gps_pos;
          // The '<=' below includes the final position if Next_SearchDist is a
          multiple of map->dist_unit.
```

```
        for( iStepNum = 0; iStepNum <= iNumSteps; iStepNum++ )
        {
            if( pmap->BoundsCheck( vStartOffset ) )
            {
                // Get the element at the front test position.
                element = WorldToMap( (long)vStartOffset.x, (long)vStartOffset.y,
        pmap);
                // Accumulate the lift.
                if( element == NULL )
                {
                    // The test point is outside the map's region.
                }
                else if( element->valid )
                {
                    lift += element->lift;
                }
                else
                {
                    // Element was inside map's region but did not have valid data.
                }

            }
            vStartOffset += vStep;
        }
        return lift;
}

float CircleSearch( Map_t *pmap, UAV_t *plane )
{
        int iAngleNum;
        float fAlpha, fStartOffset, lift;
        Vector3D vStartPos;
        std::map<float, int> LiftAngleVector;  // (lift, angle) array, sorted ascending
        by lift.
        std::map<int, float> SectorLiftVector;

        // test directly in front
        // If there is lift in front, go straight.
        // Will this allow ridge following?
        lift = CircleTest( pmap, plane, 0, Vector3D(0,0,0) );
        if( lift > 0 )
        {
            return 0.1f;
        }
        // store the lift in front
        SectorLiftVector[0] = lift;
        // test behind
        SectorLiftVector[ (int)(180 + 15 )/30.0f ] = CircleTest( pmap, plane, 180,
        Vector3D(0,0,0) );

        // Only test to the sides if there was no lift in front or behind.
        // Test the front-right & back-left quads.
        for( iAngleNum = 1; iAngleNum < Circle_NumAngles; iAngleNum++ )
        {
            fAlpha = 90 * iAngleNum / (float)Circle_NumAngles;
            fStartOffset = 2*PLANE_TURNING_CIRCLE*cos( fAlpha * DEG_TO_RAD );
            vStartPos = Vector3D( 0.0, plane->step.theta() + 90-fAlpha) *
        fStartOffset;

            lift = CircleTest( pmap, plane, fAlpha, vStartPos );
            SectorLiftVector[ (int)(( fAlpha + 15)/30.0f) ] += lift;
```

```
            lift = CircleTest( pmap, plane, fAlpha + 180, -vStartPos );
            SectorLiftVector[ (int)(( fAlpha + 180 + 15)/30.0f) ] += lift;
        }
        // test front-left & back-right quads
        for( iAngleNum = 1; iAngleNum < Circle_NumAngles; iAngleNum++ )
        {
            fAlpha =  - 90 * iAngleNum / (float)Circle_NumAngles;
            fStartOffset = 2*PLANE_TURNING_CIRCLE*cos( fAlpha * DEG_TO_RAD );
            vStartPos = Vector3D( 0.0, plane->step.theta() - fAlpha + 90) *
        fStartOffset;

            lift = CircleTest( pmap, plane, fAlpha, vStartPos );
            SectorLiftVector[ (int)(( fAlpha + 15 + 360)/30.0f) ] += lift;
            lift = CircleTest( pmap, plane, fAlpha + 180, -vStartPos );
            SectorLiftVector[ (int)(( fAlpha + 180 + 15)/30.0f) ] += lift;
        }

        // Return the sector which was best.
        // The returned angle is an offset from the UAV's current heading.
        float max = 0, min = 0;
        std::map<int, float>::iterator it = SectorLiftVector.begin(), max_it, min_it;
        while( it != SectorLiftVector.end() )
        {
            if( it->second > max )
            {
                max = it->second;
                max_it = it;
            }
            if( it->second < min )
            {
                min = it->second;
                min_it = it;
            }
            it++;
        }
        if( max == 0 )
        {
            if( min < 0 )
                return min_it->first*30 + 180;
            else
                return 0.0;
        }

        if( max_it->second <= 0.0f )
            return 0.0f;
        else
            return max_it->first*30;
}

//---------------------------------------------------------------------------
// Finds the angle that UAV needs to turn in order to head towards the specified point.
// The returned heading is in the range [0, 360).
// FlyTo90 returns true when the point has been reached.
bool FlyTo( Vector3D const &pos, Pattern_t const &Pattern, float &fHeading )
{
        float   fDist, fA, fB;
        long    longitude = Pattern.Points[2*Pattern.lCurrentPoint];
        long    latitude  = Pattern.Points[2*Pattern.lCurrentPoint+1];

        // Check if we are too close to the specified point to be able to reach it.
        // The test uses the squares of the distance so we don't have to perform
```

```
        // a sqrt() function. The comparison is valid.
        fA = (longitude - pos.x);  // east-west
        fB = (latitude - pos.y);       // north-south
        fDist = fA*fA + fB*fB; // Square of distance from plane to point.
        if( fDist <= 1.414*PLANE_TURNING_CIRCLE*PLANE_TURNING_CIRCLE )
        {
            // We are at the point
            return true;
        }

        // Point is not within turning circle. What is the angle from UAV's
        // position to the point?
        float fTheta;  // fTheta is in range [-180, 180).
        fTheta = atan2( fB, fA ) * RAD_TO_DEG;

        // Convert to heading:
        fHeading = 90 - fTheta;
        if( fHeading < 0 )
            fHeading += 360.0; // fHeading is in range [0, 360).
        return false;
}
```

Path.h

```
/*-------------------------------------------------------------------------
        The Path source files are used to search the map of VAVs and recommend
        a direction of travel in order to maximise time spent aloft.
-------------------------------------------------------------------------
File name:      path.h
Author:         Ashley Goodwin
Created:    10 April 2005
-------------------------------------------------------------------------*/
#ifndef _PATH_H_
#define _PATH_H_
/*-------------------------------------------------------------------------*/
/*      Function: TurnAdvise
        Parameters: pmap - a pointer to the map to search.
                    plane - a pointer to the UAV_t structure.
                    ang - reference to result of function, ie:
                                suggested heading.
        Returns:    integer
                    0, if successful.
                    >0, if errors.
        Purpose:
        Use current data and map data to decide which direction to turn to
        acheive maximal lift. Returns by reference the suggested heading.

        What about invalid elements?
*/
class Pattern_t
{
public:
        bool enabled;
        long *Points;
        long lNumPoints;
        long lCurrentPoint; // starting from zero
private:
        long mlGridWidth;
public:
```

```
#define lGridWidth GridWidth()
          long GridWidth(void) { return mlGridWidth; }

          bool BuildSearchPattern( Map_t *pmap );   // Are there search points yet to be
          visited?
          bool Searching( void );

          Pattern_t& operator=( Pattern_t & rh);
          Pattern_t(long width = 0, bool e = false) : Points(NULL), lNumPoints(0),
          lCurrentPoint(0), mlGridWidth(width), enabled(e) {}
          ~Pattern_t();
};

int TurnAdvise( Map_t * pmap, UAV_t *plane, float &ang );
bool FlyTo( Vector3D const &pos, Pattern_t const &Pattern, float &fHeading );

#endif
```

Sim_Map.cpp

```
          /*-------------------------------------------------------------------------
          --
              The SIM_MAP files provide simulation data for testing the UAV_t system.
              The function GetSimData() will take a position coordinate and return
              data regarding the vertical air velocity, etc. at the specified
              position.

              The returned data are generated using a simple mathematical model.
          -------------------------------------------------------------------------
          --
          File name:      sim_map.cpp
          Author:         Ashley Goodwin
          Created:        5 March 2005
          -------------------------------------------------------------------------
          */
          #include <math.h>
          #include <stdlib.h>
          #include <time.h>
          #include <string>
          #include <iostream>

          using namespace std;

          #include "trace.h"
          #include "config_file.h"
          #include "vector.h"
          #include "UAV.h"
          #include "sim_map.h"

          #define NOT_USING_LIFT_ROLLOFF
          //#define NOT_USING_FLATS

          #define max(a,b) (a>b?(a):(b))

          #define ROLL_DAMP_FACTOR   (0.7f)
          #define ROLL_LIMIT         (40.0f)

          float dist_from_point_to_segment( Vector3D , Vector3D [2]);
          float dist_to_segment_within( Vector3D P, Vector3D S[2]);
```

```
inline float dot_product( Vector3D u, Vector3D v );

Terrain_t terrain(TerrainWidth,CellWidth/CellDim);
#ifdef _DEBUG
int LiftSegment_t::num_segs(0);    // Class static variable initialisation.
#endif
/*-------------------------------------------------------------------------
--
    Wind paramters:                                                      */
namespace windspace
{
    Vector3D wind;          // Global wind vector,
    float wind_fluctuation;    //      magnitude represents wind strength
                            // Wind goes to compass point detailed by
                            //      wind angle.

}
void SetWindParam( float ws, float wa, float wf)
{
    using namespace windspace;
    // Initialise global wind vector
    wind.setangles( 0.0 , wa, ws);
    wind_fluctuation = wf;
    // Seed random number generator:
    srand( (unsigned int)time( NULL ) );
}
/*-------------------------------------------------------------------------*/
/*  Function: GetSimData
    Parameters: plane, a pointer to a UAV_t struct
                result, a pointer to a SimData_h struct where the reults are
                        stored
                deltaT, size of each Simulation-Time step.
                v_factor, the UAV_t-driven speed factor (when using straight
                        line turn approximation).
                in_lift, the amount of vertical air caused by the wind at
                        the current position. This amount is NOT for use by
                        the Turn Adivser!
    Returns:    >0, if errors occured
                0, if successful
    Purpose:
    This function serves as the source of instrumentation data, such as
    position and ground speed. It also handles the movement of the UAV_t
    within the world, governed by wind speeds & directions etc.
*/
unsigned int GetSimData( UAV_t *plane, SimData_t *result, float deltaT, float
v_factor, float &in_lift)
{
    using namespace windspace;
    long oldx = (long)plane->real_pos.x, oldy = (long)plane->real_pos.y, oldz
= (long)plane->real_pos.z;

    // TODO: remaining instruemts,

    // To find roll, get wind vector at each wing tip and find the amount of
    //  the z axis. The side with greater z amount has greater lift.
    Vector3D to_wing;
    to_wing.setangles( plane->normal.alpha(), plane->normal.theta() + 90, 0.5
* PLANE_WINGSPAN );
    if( wind.length() != 0 )
```

```
        result->Roll = ROLL_DAMP_FACTOR*90*(LocalWindVector( plane->real_pos -
to_wing, wind*deltaT, wind_fluctuation).z - LocalWindVector( plane->real_pos
+ to_wing, wind*deltaT, wind_fluctuation).z) / wind.length();
    else
        result->Roll = 0.0f;
    if( result->Roll > ROLL_LIMIT ) result->Roll = ROLL_LIMIT;
    if( result->Roll < -ROLL_LIMIT ) result->Roll = -ROLL_LIMIT;

    // Move to next position.
    // Calculate the UAV_t's new position vector, ignoring wind
    result->position = plane->real_pos + plane->step*v_factor*deltaT;
    // find local wind.
    Vector3D localwind = LocalWindVector( plane->real_pos, wind,
wind_fluctuation) * deltaT;
    result->position += localwind;
    in_lift = localwind.z;

    // Calculate instantaneous ground speed & lift
    result->Lift = (long)result->position.z - oldz;
    oldx -= (long)result->position.x;
    oldy -= (long)result->position.y;
    result->gnd_speed = (long)sqrt( oldx*oldx + oldy*oldy);
    return 0;
}
/*---------------------------------------------------------------------------*/
/*  Function: FetchTerrain
    Parameters: strfile, a refernce to a string holding the path & filename
                        of the file who's contents describe the terrain.
                others, the remaining arguments are refrences to variables
                        wi=hich will hold the read in settings.
    Returns:    bool, true if successful, false otherwise.
    Purpose:
    Reads the specified file and uses the information within to build at
    run-time the terrain of the environment, aswell as simulation settings.
*/
bool FetchTerrain( string &strfile, float &WINDSpeed, float &WINDAngle, float
&WINDFluctuation,
            long &UAVFloor, long &UAVCeiling, float &UAVStartHeading, long
&UAVStartX, long &UAVStartY, long &UAVStartZ, long &UAVGPSDelay,
            long &MAPWidth, long &MAPHeight, long &MAPLongitude, long
&MAPLatitude, long &MAPSeparation, long &MAPSearchWidth, bool
&MAPSearchEnabled, long &MAPEscapeDistance )
{
    LOG("Building terrain:");
    bool success;
    // attempt to open config file
    Config_file config_file(strfile, success);
    if( !success )
    {
        cerr << "Unable to open terrain file " << strfile << endl;
        return false;
    }

    // fetch parameters
    WINDSpeed       = config_file.SingleFloat("WINDSpeed");
    WINDAngle       = config_file.SingleFloat("WINDAngle");
    WINDFluctuation  = config_file.SingleFloat("WINDFluctuation");
    UAVFloor        = config_file.SingleLong ("UAVFloor");
    UAVCeiling      = config_file.SingleLong ("UAVCeiling");
    UAVStartHeading  = config_file.SingleFloat("UAVStartHeading");
    UAVStartX       = config_file.SingleLong ("UAVStartX");
```

```
    UAVStartY       = config_file.SingleLong ("UAVStartY");
    UAVStartZ       = config_file.SingleLong ("UAVStartZ");
    UAVGPSDelay     = config_file.SingleLong ("UAVGPSDelay");
    MAPWidth        = config_file.SingleLong ("MAPWidth");
    MAPHeight       = config_file.SingleLong ("MAPHeight");
    MAPLongitude    = config_file.SingleLong ("MAPLongitude");
    MAPLatitude     = config_file.SingleLong ("MAPLatitude");
    MAPSeparation   = config_file.SingleLong ("MAPSeparation");
    MAPSearchWidth  = config_file.SingleLong ("MAPSearchWidth");
    MAPSearchEnabled = config_file.SingleBool ("MAPSearchEnabled");
    MAPEscapeDistance= config_file.SingleLong ("MAPEscapeDistance");
    MAPEscapeDistance *= MAPEscapeDistance;

    int num_ridges, count = 0;
    config_file.get_value("num_ridges", num_ridges );
    LOG( "\tFound " << num_ridges << " ridge(s)" );
    //for each ridge..
    while( count < num_ridges )
    {
        string s = string("ridge");
        char buf[100];
        itoa( count + 1, buf, 10 );
        s.append( buf );
        LOG("\t" << s << "...");
        config_file.find_new_block( s );
        Vector3D ridge_origin;
        long angle, length;
        double rise_angle, fall_angle, left_smoothen, right_smoothen;
        long plateau_elevation, valley_elevation, plateau_length,
neutral_elevation;

        ridge_origin.x =
config_file.find_next_attr_value("ridge_origin_x").values[0].float_val;
        ridge_origin.y =
config_file.find_next_attr_value("ridge_origin_y").values[0].float_val;
        angle = (long)
config_file.find_next_attr_value("ridge_angle").values[0].float_val;
        length = (long)
config_file.find_next_attr_value("ridge_length").values[0].float_val;
        left_smoothen =
config_file.find_next_attr_value("left_smoothen_percent").values[0].float_val
;
        right_smoothen =
config_file.find_next_attr_value("right_smoothen_percent").values[0].float_va
l;

        rise_angle =
config_file.find_next_attr_value("rise_angle").values[0].float_val;
        fall_angle =
config_file.find_next_attr_value("fall_angle").values[0].float_val;
        plateau_elevation = (long)
config_file.find_next_attr_value("plateau_elevation").values[0].float_val;
        valley_elevation = (long)
config_file.find_next_attr_value("valley_elevation").values[0].float_val;
        plateau_length = (long)
config_file.find_next_attr_value("plateau_length").values[0].float_val;
        neutral_elevation = (long)
config_file.find_next_attr_value("neutral_elevation").values[0].float_val;


        fall_angle *= -1;
```

```
        // now process parameters to get terrain elevations
        long rise_length = (long)(plateau_elevation /
tan(DEG_TO_RAD*rise_angle));
        long fall_length = (long)((plateau_elevation - valley_elevation) / tan
( DEG_TO_RAD * -fall_angle ));

        // build the side-step vector...
        Vector3D side_step( cos( DEG_TO_RAD*angle ), sin( DEG_TO_RAD*angle ),
0 );
        side_step = side_step / (float)(max( fabs(side_step.x),
fabs(side_step.y) ));  // at least one component has unit length
        // build the back-step vector...
        Vector3D back_step( cos( DEG_TO_RAD*(90+angle) ), sin(
DEG_TO_RAD*(90+angle) ), 0 );
        back_step = back_step / max( fabsf(back_step.x), fabsf(back_step.y) );


        // move sideways on ridge (far end to far end)
        Vector3D slice_origin =ridge_origin, ridge_end =ridge_origin +
Vector3D( length*cos( DEG_TO_RAD*angle ), length*sin( DEG_TO_RAD*angle ), 0
);
        int num_steps = (int)(length / side_step.length()), c = 0;
        //while( slice_origin != (ridge_end + side_step ) )
        while( c <= num_steps )
        {
            // move front to back of ridge
            Vector3D point = slice_origin;
            float dist_from_slice_origin = 0;
            while( terrain.bounds_check( point.x, point.y) )
            {   //moving front to back now...
                double h=0,step_dist = back_step.length();;
                static oldh=0;

                if( dist_from_slice_origin < rise_length )
                {
                    // rise slope
                    h = dist_from_slice_origin*tan(DEG_TO_RAD*rise_angle);
                }
                else if( dist_from_slice_origin < rise_length + plateau_length
)
                {
                    // plateau
                    h = plateau_elevation;
                }
                else if( dist_from_slice_origin < rise_length + plateau_length
+ fall_length )
                {
                    // fall slope
                    h = valley_elevation + (rise_length + plateau_length +
fall_length - dist_from_slice_origin)*tan( DEG_TO_RAD * -fall_angle );
                }
                else
                    h = valley_elevation;  // valley

                // smoothen edges
                float p = c / (float)num_steps;
                if( p < left_smoothen )
                    h *= (p / left_smoothen);
                else if( p > (1-right_smoothen) )
                    h *=  (1 - p)  / right_smoothen;
```

```
                // store height (h) into array;
                //terrain.altitude( point.x, point.y ) = (float)h;
                terrain.altitude( point.x, point.y, (float)h );
                terrain.neutral_altitude(point.x, point.y ) =
(float)neutral_elevation;
                if( h > terrain.max_altitude() ) terrain.max_altitude() =
(float)h;

                point += back_step;
                dist_from_slice_origin += (float)step_dist;
            }
            slice_origin += side_step;
            c++;
        }

        // now build the lift segment correspoding to this ridge
        // rise segment
        Vector3D rise_origin, fall_origin, fwd;
        double m_far, m_near, o;

        Vector3D to_rise_origin;
        to_rise_origin.setangles( rise_angle, -angle, neutral_elevation /
sin(DEG_TO_RAD*rise_angle) );
        rise_origin = ridge_origin + to_rise_origin;


        fwd = Vector3D(ridge_origin.x - rise_origin.x,ridge_origin.y -
rise_origin.y, 0 );
        m_far = (rise_origin - ridge_origin).length();
        m_near = (float)(m_far - plateau_elevation / sin(
DEG_TO_RAD*rise_angle ));

        terrain.add_segment( LiftSegment_t(
                                rise_origin, rise_origin + side_step.normal() *
length,
                                fwd, LiftSegment_t::RISE, m_near, m_far,
                                neutral_elevation / (float)(neutral_elevation -
plateau_elevation),
                                left_smoothen, right_smoothen ) );
        // fall segment
        // 'o' holds the horizontal offset front-to-back along the ridge to
the fall origin
        o = plateau_elevation / tan(DEG_TO_RAD*rise_angle) +
                plateau_length - (neutral_elevation - plateau_elevation ) /
tan(-DEG_TO_RAD*fall_angle);

        fall_origin = ridge_origin + Vector3D(  -sin(DEG_TO_RAD*angle) * o,
                                                cos(DEG_TO_RAD*angle) * o,
                                                neutral_elevation
                                                );
        fwd *= -1;
        m_near = (neutral_elevation - plateau_elevation) / sin(-
DEG_TO_RAD*fall_angle);
        m_far =  (neutral_elevation - valley_elevation ) / sin(-
DEG_TO_RAD*fall_angle);

        terrain.add_segment( LiftSegment_t(
                                fall_origin, fall_origin + side_step.normal() *
length,
                                fwd, LiftSegment_t::FALL, m_near, m_far,
```

```
                                neutral_elevation / (float)(neutral_elevation -
valley_elevation),
                                left_smoothen, right_smoothen ) );
        // do next ridge
        count++;
        LOG("\t\t...built.");
    }
#ifndef NOT_USING_FLATS
    // get added flats
    int num_flats;
    count=0;
    config_file.get_value_assert("num_flats", num_flats);
    while( count < num_flats )
    {
        long x1,x2,x3,x4,y1,y2,y3,y4;
        string s = string("flat");
        char buf[100];
        itoa( count + 1, buf, 10 );
        s.append( buf );
        LOG("\t" << s << "...");
        config_file.find_new_block( s );

        x1 = config_file.find_next_attr_value("x1").values[0].float_val;
        y1 = config_file.find_next_attr_value("y1").values[0].float_val;
        x2 = config_file.find_next_attr_value("x2").values[0].float_val;
        y2 = config_file.find_next_attr_value("y2").values[0].float_val;
        x3 = config_file.find_next_attr_value("x3").values[0].float_val;
        y3 = config_file.find_next_attr_value("y3").values[0].float_val;
        x4 = config_file.find_next_attr_value("x4").values[0].float_val;
        y4 = config_file.find_next_attr_value("y4").values[0].float_val;

        Vector3D step_14, step_23;
        // calculate start & end points for line 1-4 and calculate step of
(x,y) to get there
        // do similar for 2-3 line
        float mx = (float)max( max( abs(x4 - x1), abs(y4 - y1)),
                               max( abs(x3 - x2), abs(y3 - y2)) );
        step_14 = Vector3D( (x4-x1) / mx, (y4-y1) / mx, 0);
        step_23 = Vector3D( (x3-x2) / mx, (y3-y2) / mx , 0);

        Vector3D point_14(x1, y1, 0), point_23(x2, y2, 0);
        Vector3D p4(x4, y4, 0);
        Vector3D p1(x1, y1, 0);

        float alt_1 = terrain.altitude( (float)x1, (float)y1 );
        float alt_2 = terrain.altitude( (float)x2, (float)y2 );
        float diff_14 = terrain.altitude( (float)x4, (float)y4 ) - alt_1;
        float diff_23 = terrain.altitude( (float)x3, (float)y3 ) - alt_2;

        int num_steps_14 = (int)mx;
        int count_14 = 0;
        while( count_14 <= num_steps_14 )
        {
            // calculate value at this point along 14 & 23 lines as linear
interpolation
            // between 1&4 and 2&3.
            terrain.altitude( point_14.x, point_14.y ) = alt_1 +
(count_14/(float)num_steps_14) * diff_14;
            terrain.altitude( point_23.x, point_23.y ) = alt_2 +
(count_14/(float)num_steps_14) * diff_23;
```

```
            //for each pair of points an i-th along the lines, interpolate
line between two points.
            mx = (float)(max( fabs(point_23.x - point_14.x), fabs(point_23.y -
point_14.y)));
            Vector3D step_14_23 = Vector3D( (point_23.x - point_14.x) / mx,
(point_23.y - point_14.y) / mx, 0);

            Vector3D point_14_23 = point_14;
            //float diff_14_23 = terrain.altitude(point_23.x, point_23.y) -
terrain.altitude( point_14.x, point_14.y);
            int num_steps_14_23 = (int)mx;
            int count_14_23 = 0;
            while( count_14_23 <= num_steps_14_23 )
            {
                // terrain at point_14_23 = interpolation between point_14 and
point_23
                float diff_14_23 = (point_14_23 - point_14).length() /
(point_23 - point_14).length();

                // add flats into the equation
                terrain.altitude( point_14_23.x, point_14_23.y) =
terrain.altitude( point_14.x, point_14.y) +
(count_14_23/(float)num_steps_14_23) * diff_14_23;

                point_14_23 += step_14_23;
                count_14_23++;
            }

            point_14 += step_14;
            point_23 += step_23;
            count_14++;
        }
        count++;
        LOG("\t\t...built");
    }
#endif
    LOG("Finished building terrain.");
    return true;
}
/*----------------------------------------------------------------------------*/
/*  Function: LocalWindVector
    Parameters: pos, a vecotr indicating the UAV_t's current position
                speed_factor, a float in which the speed factor will be
                    returned.
                wind, a vector specifiying the global wind.
                wind_fluc, a float describing the percentage wind
                    strength fluctuates by.
    Returns:    a vector, indicating the direction of the local wind
    Purpose:
    The function calculates in which direction the wind is travelling at
    the specified position. If wind direction is unaffected by the terrain,
    it returns the global wind.
*/
Vector3D LocalWindVector( Vector3D pos, Vector3D &wind, float wind_fluc )
{
    float d, min=-1, speed_factor;
    float fluc_factor;

    // Get a random fluctuation factor:
    fluc_factor = 1 - 2 * wind_fluc * ( (rand() / (float)RAND_MAX) - 0.5);
```

```
            // The above gets a number in range [1 - wind_fluc, 1 + wind_fluc]

    // set default speed factor
    if( terrain.neutral_altitude( pos.x, pos.y ) == 0 )
        speed_factor = 1;
    else
        speed_factor = terrain.neutral_altitude( pos.x, pos.y ) /
(terrain.neutral_altitude( pos.x, pos.y ) - terrain.altitude( pos.x, pos.y
));

    // determine which lift segment is closest and is valid
    vector<LiftSegment_t>::iterator min_itr, itr = terrain.m_lift.begin();

    // check all segments
    while( itr != terrain.m_lift.end() )
    {
        // get forward vector
        Vector3D fwd = itr->forward.normal();

        // get 'flat vector' from closest point on segment to pos  (no z
component)
        Vector3D to_point = pos - itr->endpoint[0];
        Vector3D line = itr->endpoint[1] - itr->endpoint[0];
        d = dot_product( to_point.normal(), line.normal() ) ;
        Vector3D perp_point = itr->endpoint[0] + line.normal() *
to_point.length() * d;
        Vector3D flat = (pos - perp_point).normal();
        flat.z = 0;

        d = dist_to_segment_within( pos, itr->endpoint );
        // if dot product is positive, segment MAY be valid
        // if distance to segment is within the ranges, segment MAY be valid
        // if current UAV_t position is below the lift segment, segment MAY be
valid
        // need ALL THREE conditions to be true for segment to be valid
        if( dot_product( fwd, flat ) >= 0 &&  d > itr->near_limit && d < itr-
>far_limit && pos.z < itr->endpoint[0].z )
        {
            // valid, remember it
            if( min == -1 )
            {
                min = d;
                min_itr = itr;
            }
            else if( d < min )
            {
                // keep the closest segment
                min_itr = itr;
            }
        }
        // check out next segment...
        itr++;
    }
    // no segments valid?
    if( min == -1 ) return wind * speed_factor * fluc_factor;
    LiftSegment_t ls = *min_itr;

    // ** Calculate angle of wind based on lift segment ** \\

    // this is the line vector
    Vector3D line = ls.endpoint[1] - ls.endpoint[0];
```

```
    // this is vector from endpoint to pos
    Vector3D to_point = pos - ls.endpoint[0];
    d = dot_product( to_point.normal(), line.normal() ) ;            // get
cos( angle between vectors )


    // calculate the  vector from pos to perpendicular point
    Vector3D perp_point = ls.endpoint[0] + line.normal() * to_point.length() *
d;
    Vector3D r = perp_point - pos;

    // determine rolloff
#ifndef NOT_USING_LIFT_ROLLOFF
    float p = perp_point.length();
    p /= ( ls.endpoint[1] + ls.endpoint[0] ).length();
    if( p < ls.left_rolloff )
        r.z *= (p / ls.left_rolloff);
    else if( p > (1-ls.right_rolloff) )
        r.z *=  (1 - p)  / ls.right_rolloff;
#endif

    // Adjust amount of vertical movement according to the angle the wind
    // makes with the ridge. When they are perpendicular, there is no vertical
movement.
    Vector3D local;
    float factor = sin( DEG_TO_RAD *  fabs((int)line.theta() -
(int)wind.theta()) );
    local.setangles( r.alpha() * factor, wind.theta(), wind.length() );

    // is it a rise or fall segment?
    if( ls.type == LiftSegment_t::FALL )
        local *= -1;

    // calculate speed factor
    speed_factor = ( r.length() - ls.far_limit ) * ( 1 - ls.near_speed_factor)
/ ( ls.far_limit - ls.near_limit ) + 1;

    return local.normal() * wind.length() * speed_factor * fluc_factor;
    // i.e. the direction of local, and the speed of ( wind speed *
spped_factor * fluc_factor )
}
/*---------------------------------------------------------------------------*/
/*  Function: dot_product
    Parameters: two vectors
    Returns:    float
    Purpose:
    Returns the dot product of the two vectors passed.
*/
inline float dot_product( Vector3D u, Vector3D v )
{
    return u.x * v.x + u.y * v.y + u.z * v.z;
}

/*---------------------------------------------------------------------------*/
/*  Function: dist_from_point_to_segment
    Parameters: P, the test point
                S, an array of two vector specifying the endpoints of the
                    segment.
    Returns:    float
    Purpose:
    Returns the distance from the test point to the segment.
```

```
    This function was originally taken from the webpage:
    'About Lines and Distance of a Point to a Line (2D & 3D)'
    authored by Dan Sunday
    (http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm).

    It was modified by Ashley Goodwin to suit the purposes of this program.
*/
float dist_from_point_to_segment( Vector3D P, Vector3D S[2])
{
    Vector3D v = S[1] - S[0];
    Vector3D w = P - S[0];

    double c1 = dot_product(w,v);
    if ( c1 <= 0 )
        return (P - S[0]).length();

    double c2 = dot_product(v,v);
    if ( c2 <= c1 )
        return (P - S[1]).length();

    double b = c1 / c2;
    Vector3D Pb = S[0] + v * b;
    return (P - Pb).length();
}
/*---------------------------------------------------------------------------*/
/*  Function: dist_to_segment_within
    Parameters: P, the test point
                S, an array of two vector specifying the endpoints of the
                    segment.
    Returns:    float
    Purpose:
    Returns the distance from the test point to the segment, unless the
    point lies outside of the cylinder surrounding S. If this is the case,
    it returns -1.

    This function was originally taken from the webpage:
    'About Lines and Distance of a Point to a Line (2D & 3D)'
    authored by Dan Sunday
    (http://softsurfer.com/Archive/algorithm_0102/algorithm_0102.htm).

    It was modified by Ashley Goodwin to suit the purposes of this program.
*/
float dist_to_segment_within( Vector3D P, Vector3D S[2])
{
    Vector3D v = S[1] - S[0];
    Vector3D w = P - S[0];

    double c1 = dot_product(w,v);
    if ( c1 <= 0 )
        return -1;

    double c2 = dot_product(v,v);
    if ( c2 <= c1 )
        return -1;

    double b = c1 / c2;
    Vector3D Pb = S[0] + v * b;
    return (P - Pb).length();
}
```

```
//-------------------------------------------------------------------------
---
//          Terrain Class function definitions
//
Terrain_t::Terrain_t(long w, float d) : width(w), dist_between_indices(d),
max_elevation(0)
{
    // Allocate memeory to store terrain altitudes
    w = (long)( Terrain_t::width / Terrain_t::dist_between_indices );
    map = new struct internal *[w+2];
    if( !map )
        exit(2);
    for( int i = 0; i < w; i ++ )
    {
        map[i] = new struct internal[w+2];
        if( !map[i] )
        {
            // free any mem allocated before this allocation failed
            i--;
            for( ; i >= 0; i-- )
                delete [] map[i];

            exit(2);
        }
        memset( map[i], 0, sizeof(struct internal) * (w+1) );
    }
}
/*-------------------------------------------------------------------------*/
float& Terrain_t::altitude(float i, float j)  // use the parameters to access
element of array
{
    if( i >= Terrain_t::width ) i = (float)Terrain_t::width-1;
    if( i <= 0 )    i = 1;
    if( j >= width ) j = (float)width-1;
    if( j <= 0 )    j = 1;
    return
map[(long)(i/dist_between_indices)][(long)(j/dist_between_indices)].elevation
;
}
/*-------------------------------------------------------------------------*/
// Sets altitude to e and returns reference.
float& Terrain_t::altitude(float i, float j, float e)     // use the parameters
to access element of array
{
    if( i >= Terrain_t::width ) i = (float)Terrain_t::width-1;
    if( i <= 0 )    i = 1;
    if( j >= width ) j = (float)width-1;
    if( j <= 0 )    j = 1;

    float &alt =
map[(long)(i/dist_between_indices)][(long)(j/dist_between_indices)].elevation
;

    if( alt != 0.0 )
        return alt;

    alt = e;
    return ( alt );
}
/*-------------------------------------------------------------------------*/
```

```
float& Terrain_t::neutral_altitude( float i, float j) // Return the neutral
elevation.
{
    if( i >= width ) i = (float)width-1;
    if( i <= 0 )     i = 1;
    if( j >= width ) j = (float)width-1;
    if( j <= 0 )     j = 1;
    return
map[(int)(i/dist_between_indices)][(int)(j/dist_between_indices)].straight_th
rough_altitude;
}
/*---------------------------------------------------------------------------*/
LiftSegment_t Terrain_t::lift( long i ) // Gives access to the lift segments
{
    return m_lift[i];
}
/*---------------------------------------------------------------------------*/
void Terrain_t::add_segment( LiftSegment_t &ls ) // Adds a lift segment
{
    m_lift.push_back( ls );
}
/*---------------------------------------------------------------------------*/
float& Terrain_t::max_altitude( void )
{
    return max_elevation;
}
// bounds_check:
// checks whether the passed x & y coordiantes are within the range
// for which terrain data is held.
inline bool Terrain_t::bounds_check( long x, long y)
{
    return ( x >= 0 && x <= Terrain_t::width && y >= 0 && y <=
Terrain_t::width );
}
/*---------------------------------------------------------------------------*/
Terrain_t::~Terrain_t()    //destructor
{
    //free allocated memeory...
    long w = (long)(Terrain_t::width / Terrain_t::dist_between_indices);
    for( int i = 0; i < w; i ++ )
    {
        delete [] map[i];
    }
    delete [] map;
}
// END of Terrain_t function definitions.
/*---------------------------------------------------------------------------*/
```

Sim_Map.h

```
/*-----------------------------------------------------------------------------
        The SIM_MAP files provide simulation data for testing the UAV_t system.
        The function GetSimData() will take a position coordinate and return
        data regarding the vertical air velocity, etc. at the specified
        position.

        The returned data are generated using a simple model.
-------------------------------------------------------------------------------
--
File name:      sim_map.h
```

```
Author:         Ashley Goodwin
Created:        5 March 2005
-------------------------------------------------------------------------
*/
#ifndef _SIM_MAP_H_
#define _SIM_MAP_H_

#include <GL\glut.h>
#include <vector>      // This file uses the STL Vector
using namespace std;


#ifndef VECTOR_H       // This file requires vectors
#include "vector.h"
#endif

/* COnfiguration parameters: */
#define TerrainWidth    (6000)  // terrain is a square of this width*width
#define NumCells        (20)        /* Terrain grid is NumCells X NumCells
cells */
#define CellDim         (6)         /* Each cell is CellDim X Celldim quads */
#define CellWidth       (TerrainWidth/NumCells)

/* Class & struct declarations.        */
typedef struct {
        unsigned int    *flags;    // stores flags for the cell.
        float           midz;   // stores the middle z value of the cell.
        GLfloat         *vdata;    // stores vertex info.
        GLfloat         *tdata; // stores texture coordinate info.
        GLfloat         *cdata; // stores colour information for each vertex.
        GLfloat         *ndata; // stores normal vector information for each
quad
} DrawObj_t;

class LiftSegment_t
{
#ifdef _DEBUG
public:
    static int num_segs;
    int my_seg_num;
#endif
public:
    enum LiftType{ RISE, FALL };

    Vector3D    endpoint[2];
    LiftType    type;
    Vector3D    forward;                // points to area where lift is valid
    float       near_limit, far_limit; // range of radii lift resides in
    float       near_speed_factor;
    float       left_rolloff, right_rolloff;// amount of ridge that is
smoothened, at each end

#ifdef _DEBUG
    LiftSegment_t(){my_seg_num = ++num_segs;}
#else
    LiftSegment_t() { };        // empty constructor
#endif
    LiftSegment_t( Vector3D va, Vector3D vb, Vector3D vf, LiftType lt, double
nl, double fl, double nsf, double left_rlf, double right_rlf )
        :
        forward(vf),
```

```
        type(lt),
        near_limit((float)nl),
        far_limit((float)fl),
        near_speed_factor((float)nsf),
        left_rolloff((float) left_rlf),
        right_rolloff((float) right_rlf)
    {
        // other initialisation...
        endpoint[0] = va;
        endpoint[1] = vb;
#ifdef _DEBUG
        my_seg_num = ++num_segs;
#endif
    }

    LiftSegment_t& operator=(LiftSegment_t ls )
    {
        endpoint[0] = ls.endpoint[0];
        endpoint[1] = ls.endpoint[1];
        type = ls.type;
        forward = ls.forward;
        near_limit = ls.near_limit;
        far_limit = ls.far_limit;
        near_speed_factor = ls.near_speed_factor;
        left_rolloff = ls.left_rolloff;
        right_rolloff = ls.right_rolloff;
        return *this;
    }

};

class Terrain_t
{
private:
    struct internal { float elevation,straight_through_altitude; };  // each
point of the terrain is of this type
    float max_elevation;
    vector <LiftSegment_t> m_lift;        // holds the lift segements in a STL
vector
    struct internal **map;                // map of terrain data, type 'struct
internal'
public:
    DrawObj_t       **cells;              // cells containing terrain vertices
    const long     width;                 // width of terrain stored
    const float    dist_between_indices;  // distance between each index in
map

    Terrain_t(long w = TerrainWidth, float d = (1) );
    ~Terrain_t();

    float& altitude(float i, float j);
    float& altitude(float i, float j, float e);
    float& neutral_altitude( float i, float j);
    LiftSegment_t lift( long i );
    void add_segment( LiftSegment_t &ls );
    float& max_altitude( void );

    bool bounds_check( long x, long y);

    friend Vector3D LocalWindVector( Vector3D pos, Vector3D &wind, float
wind_fluc );
```

```
    friend void Display(void);

};

struct SimData_t   // returned by reference by the GetSimData() fn.
{
    long       Lift;       // Altitude rise/fall of UAV_t
    float      Roll;       // Roll of the UAV_t, +ve clockwise when
                           //  looking in direction of travel.
    long       gnd_speed;  // Ground speed meausred in m/s.
    Vector3D   position;   // The posiition of the plane.

};

// Function declarations.
unsigned int GetSimData( UAV_t *plane, SimData_t *result, float deltaT, float
v_factor, float &in_lift );
void SetWindParam( float ws, float wa, float fluctuation );
bool FetchTerrain( string &strfile, float &WINDSpeed, float &WINDAngle, float
&WINDFluctuation,
            long &UAVFloor, long &UAVCeiling, float &UAVStartHeading, long
&UAVStartX, long &UAVStartY, long &UAVStartZ, long &UAVGPSDelay,
            long &MAPWidth, long &MAPHeight, long &MAPLongitude, long
&MAPLatitude, long &MAPSeparation, long &MAPSearchWidth, bool
&MAPSearchEnabled, long &MAPEscapeDistance );

inline float dot_product( Vector3D u, Vector3D v );

#endif
```

Trace.cpp

```
#include <string>
#include <stdlib.h>
using namespace std;

#define _TRACE_CPP_
#include "trace.h"

// create tracers
Trace_t TraceFile("log.csv");
Trace_t TraceScreen;

// Turn tracing on/off
void Tracing(bool b)
{
    TraceScreen.trace_enabled = b; TraceFile.trace_enabled = b;
}
// Turn logging on/off
void Logging(bool b)
{
    TraceScreen.log_enabled = b; TraceFile.log_enabled = b;
}
//------------------------------------------------------------------------
ostream& Trace_t::Log( void )
{
    if( to_file)
        return file;
    else
        return cout;
```

```
}
//---------------------------------------------------------------------------
// Output to file...
Trace_t::Trace_t(string outfile) : file( outfile.c_str() ) , to_file(true),
trace_enabled(false), log_enabled(true)
{
    if( !file )
    {
        //error
        cerr << "Couldn't open log file. Exiting." << endl;
        exit(1);
    }
}
// Output to cout.
Trace_t::Trace_t() : file(), to_file(false), trace_enabled(false),
log_enabled(true)
{
}
//---------------------------------------------------------------------------
// Destructor
Trace_t::~Trace_t()
{
    if( to_file )
        file.close();
    else
        cout.flush();
}
//---------------------------------------------------------------------------
// Output the passed data.
Trace_t& Trace_t::operator<<( string s )
{
    if( !trace_enabled ) return *this;

    if( to_file)
    {
        file << s;
        file.flush();;
    }
    else
    {
        cout << s;
        cout.flush();
    }

    return *this;
}
Trace_t& Trace_t::operator <<(char c )
{
    char buffer[]={c,'\0'};
    return *this << string(buffer);
}
Trace_t& Trace_t::operator<<( long i )
{
    char buffer[34];
    ltoa(i, buffer, 10);
    return *this << buffer;
}
```

Trace.h

```
#ifndef _TRACE_H_
          #define _TRACE_H_

          #include <iostream>
          #include <fstream>
          #include <string>

          class Trace_t
          {
          private:
              std::ofstream file;
              bool to_file;
          public:
              bool trace_enabled;
              bool log_enabled;
              std::ostream& Log( void );

              Trace_t& operator<<( std::string s );
              Trace_t& operator<<( char c );
              Trace_t& operator<<( long i );
              Trace_t& operator<<( int i )   { return (*this << (long)i); }
              Trace_t& operator<<( char *s ) { return (*this << std::string(s)); }
              Trace_t(std::string outfile);
              Trace_t();
              ~Trace_t();
          };

          // make reference to tracers
          #ifndef _TRACE_CPP_
              extern Trace_t TraceFile;
              extern Trace_t TraceScreen;
          #endif // _TRACE_CPP_

          // TRACE only works when debugging.
          #define TRACE(s)
          #if defined(_DEBUG)

          #undef TRACE
          #define TRACE(s) TraceScreen << s << '\n'

          #endif  // _DEBUG

          // Assert _always_ writes to cout if condition e is true
          #define ASSERT(e,s) if(e) TraceScreen.Log() << "File: " << __FILE__ << "
          Line: " << __LINE__ << ":" << s << '\n'; TraceScreen.Log().flush()
          #define LOG(s) if(TraceScreen.log_enabled) TraceScreen.Log() << s << '\n';
          TraceScreen.Log().flush()
          #define LOGFILE(s) if(TraceFile.log_enabled) TraceFile.Log() << s << '\n';
          TraceFile.Log().flush()

          void Tracing(bool b);
          void Logging(bool b);

          #endif  // _TRACE_H_
```

UAV.h

```
/*-----------------------------------------------------------------------------
              The UAV.h file defines the UAV struct aswell as describing the physical
```

```
    structure of the plane.
--------------------------------------------------------------------------------
--
File name:      UAV.h
Author:         Ashley Goodwin
Created:        22 April, 2005
--------------------------------------------------------------------------------
*/
#ifndef _UAH_H_
#define _UAV_H_
#include "vector.h" // This files requires vectors

class UAV_t
{
public:
    float       air_speed;          // Air speed of the aircraft, measured in
m/s.
    Vector3D    normal;             // Roll, pitch & yaw of UAV.
    Vector3D    step;               // Amount to move each timestep. I.e.,
                                    //          new = old + step;
    Vector3D    real_pos;           // Current ACTUAL position, in metres.
    Vector3D    gps_pos;            // Position of UAV as reported by GPS

    float       roll, roll_target; // The amount of roll the UAV has, and the
amount of roll
                                    //      it wants.
    long        floor, ceiling;     // The minimum altitude the UAV should try
to keep.
    float       gps_delay;          // The amount of time the GPS takes to
report a position (secs).
    int         heading_target;
};
/* The plane:
          W
          |
      T---C--N
          |
          W

  where T=Tail
        N=Nose
        W=wingtip
*/
#define PLANE_WINGSPAN          (0.8f)      // metres, tip to tip
#define PLANE_LENGTH            (0.6f)
#define PLANE_HEIGHT            (0.2f)
#define PLANE_WIDTH                 (0.12f)
#define PLANE_WING_HEIGHT       (0.03f)
#define PLANE_WING_WIDTH        (0.15f)
#define PLANE_RUDDER_HEIGHT         (0.1f)
#define PLANE_RUDDER_WIDTH      (0.015f)
#define PLANE_RUDDER_LENGTH         (0.1f)
#define PLANE_CENTRE            (0.2f)      // Dist from nose to centre axis of
wings (marked C above)
#define PLANE_BOUNDING_RADIUS   (0.8f)

#define ROLL_ADJUST_PER_STEP    (5.0f)      // Maximum degrees per unit time
step that roll can be moved
#define PITCH_ADJUST_PER_STEP   (15.0f)     // Maximum degrees to nose up/down
to change pitch.
```

```
#define PLANE_LEVEL_ALPHA      (28.5f)     // This angle offsets the effects of
gravity and allows
                                           // the UAV to fly levelly.
                                           // = atan( 9.8 / CRUISE_SPEED )
(degrees)

#define PLANE_CRUISE_SPEED     (20.0f)     // Cruise speed, m/s
#define PLANE_TURNING_CIRCLE   (30.f)      // Turning circle radius @ cruise
speed
// BETA_NOUGHT: Constant describing the tightness of the equivilant straight
line turn around a
//              turning circle during a one second time step.
//              It is a function of UAV cruise speed and turning circle radius
at
//              cruising speed.
/*
    Suav = cruising speed of UAV (assumed to remain constant )
    Rtc = radius of turning circle

    beta_nought := 0.5 * (360degrees) * Suav / (2*PI*Rtc);
  */
#define BETA_NOUGHT    (0.5 * 360 * PLANE_CRUISE_SPEED /
(2*M_PI*PLANE_TURNING_CIRCLE) )

#endif
```

Vector.h

```
/*-----------------------------------------------------------------------------
    This file creates a 3 dimensional vector class and defines a number of
    operators and functions that can be performed on a vector.
-------------------------------------------------------------------------------
--
File name:     vector.h
Author:        Ashley Goodwin
-------------------------------------------------------------------------------
*/
#ifndef _VECTOR_H_
#define _VECTOR_H_

#include <GL\glut.h>

#ifndef M_PI
#define M_PI            ((double)(3.14159265358979323846))
#endif

#ifndef RAD_TO_DEG
#define RAD_TO_DEG     (180.0/M_PI)
#endif

#ifndef DEG_TO_RAD
#define DEG_TO_RAD     (M_PI/(double)180.0)
#endif

#include <iostream>
#include <math.h>

class Vector3D
{
public:
```

```
    GLfloat x, y, z;
    // Constructors
    Vector3D() : x(1), y(0), z(0)
    {}
    Vector3D(double xx, double yy, double zz ) : x((float)xx), y((float)yy),
z((float)zz)
    {}
    Vector3D(float alpha, float theta) : x(1),y(0),z(0)
    {
        // Vectors created this way have unit length.
        setangles( alpha, theta, 1 );
    }

    Vector3D& operator=( Vector3D r)   // Copy one vector to another
    {
        x = r.x;
        y = r.y;
        z = r.z;
        return *this;
    }
    Vector3D operator+ (Vector3D r)    // add two vectors
    {
        return Vector3D(x + r.x, y + r.y, z + r.z);
    }
    Vector3D operator- (Vector3D r)    // subtract two vectors
    {
        return Vector3D(x - r.x, y - r.y, z - r.z);
    }
    Vector3D operator* (double val)    // scale up a vector
    {
        return Vector3D(x * val, y * val, z * val);
    }
    Vector3D operator/ (float val) // scale down a vector
    {
        return Vector3D(x / val, y / val, z / val);
    }
    Vector3D& operator+= (Vector3D r)  // add vector to this one
    {
        x += r.x;
        y += r.y;
        z += r.z;
        return *this;
    }
    Vector3D& operator-= (Vector3D r)  // minus a vector from this one
    {
        x -= r.x;
        y -= r.y;
        z -= r.z;
        return *this;
    }
    Vector3D& operator*= (float val)   // scale the vector up
    {
        x *= val;
        y *= val;
        z *= val;
        return *this;
    }
    Vector3D& operator/= (float val)   // scale the vector down
    {
        x /= val;
        y /= val;
```

```
        z /= val;
        return *this;
    }
    Vector3D operator- ()  // operator- is used to set this Vector3D's x, y,
and z to the negative of them.
    {
        return Vector3D(-x, -y, -z);
    }
    bool operator==( Vector3D& v ) //equality test
    {
        return v.x == x && v.y == y && v.z == z;
    }
    bool operator!=( Vector3D& v ) //inequality test
    {
        return !( *this == v );
    }
    bool operator<=( Vector3D &v ) // compares on length
    {
        return length() <= v.length();
    }
    float length()  // length() returns the length of this Vector3D
    {
        return sqrtf(x*x + y*y + z*z);
    };

    void normalise()   // normalizes this Vector3D that its direction remains
the same but its length is 1.
    {
        float length = this->length();

        if (length == 0)
            return;

        x /= length;
        y /= length;
        z /= length;
    }
    Vector3D normal()  // returns a new Vector3D. The returned value is a
normalised version of this Vector3D.
    {
        float length = this->length();

        if (length == 0)
            return *this;

        return Vector3D(x / length, y / length, z / length);
    }
    double alpha( void )   // returns the alpha value. this is the vertical
angle.
    {
        return RAD_TO_DEG * asin( z / this->length() );
    }
    double theta( void )   // returns the theta value. this is eqv. to heading
    {                      // returned value is [0, 360)
        if( fabs(y) <= 1e-5 )      // smaller than 10^-5 is good enough for
zero
            if( x > 0 )
                return 90.0;
            else
                return -90.0;
        else
```

```
        {
            float val = RAD_TO_DEG * atan2( x, y );
            if( val < 0 ) val += 360.0f;
            return val;
        }
    }
    // alpha is vertical angle from x-y plane to z.
    // theta in CW angle from y to x.
    Vector3D& setangles( float alpha, float theta )   // make a vector from
angles.
    {
        return setangles( alpha, theta, this->length());
    }
    Vector3D& setangles( double alpha, double theta, float len )  // make a
vector from angles.
    {
        x = (float)(len * cos( alpha*DEG_TO_RAD ) * sin( theta*DEG_TO_RAD ));
        y = (float)(len * cos( alpha*DEG_TO_RAD ) * cos( theta*DEG_TO_RAD ));
        z = (float)(len * sin( alpha*DEG_TO_RAD ));
        return *this;
    }

    friend std::ostream& operator<<(std::ostream &os, Vector3D v )    // print
the vecotr's components
    {
        os << "X: " << v.x << " Y: " << v.y << " Z: " << v.z << " A: " <<
v.alpha() << " T: " << v.theta();
        return os;
    }

    Vector3D cross( Vector3D &v )  // return the cross-product of this vector
with another: THIS x ARGUMENT
    {
        return Vector3D( y*v.z - v.y*z, x*v.z - v.x*z, x*v.y - v.x*y);
    }

};

#endif
```

8.5. *Conference Paper*