1. Analysis and design
   1. Contextual inquiry

As I have been working closely with the end-users in the very lab they are going to be working with the project I felt no need for a formal contextual interview but rather I got the chance to observe how work was performed all the time. I could ask questions about routines, issues that might arise and generally observing work as it happened. This was especially relevant for the design of the interactive calibration wizard. The advanced hardware setup proved to be difficult to calibrate.

Therefore, before implementation of the calibration wizard was performed myself and the main user performed a trial calibration by manually calculating all the necessary values. Doing this before designing the calibration wizard made it easier to model the work process as natural as possible. The trial calibration gave an insight into what steps were necessary and in, more importantly, in which order.

* 1. Sequence models

Intent: Perform an experiment without trigger

Trigger: User starts program and calibration is done

* Load old calibration file
* Change settings to no trigger and plotting to the preferred plotting
* Press Run
* Press Stop when needed

Intent: Calibrate software

Trigger

* First program run after changing setup
* Measure ball radius and decide what is defined as forward in relation to the two sensors
* Identify each sensor and the desktop mouse
* Calibrate translation by moving the ball in a way that simulates translation in the fly’s coordinate plane. Also measure this distance
* Do this several times and measure average error to find and decide whether or not its acceptable
* Redo this for yaw
  1. Paper prototype

The paper prototype was made in two iterations before the actual draft of a GUI was first implemented. Using a combination of the open source tool Pencil and actual drawings a prototype was produced with no interaction for the user but clearly specified what actions are possible and what they lead to. This was then displayed for the end users and they had a chance to give feedback.

The prototype for the calibration step is displayed in figure 1 to 3:



Figure 1 - The calibration wizard (1/2)

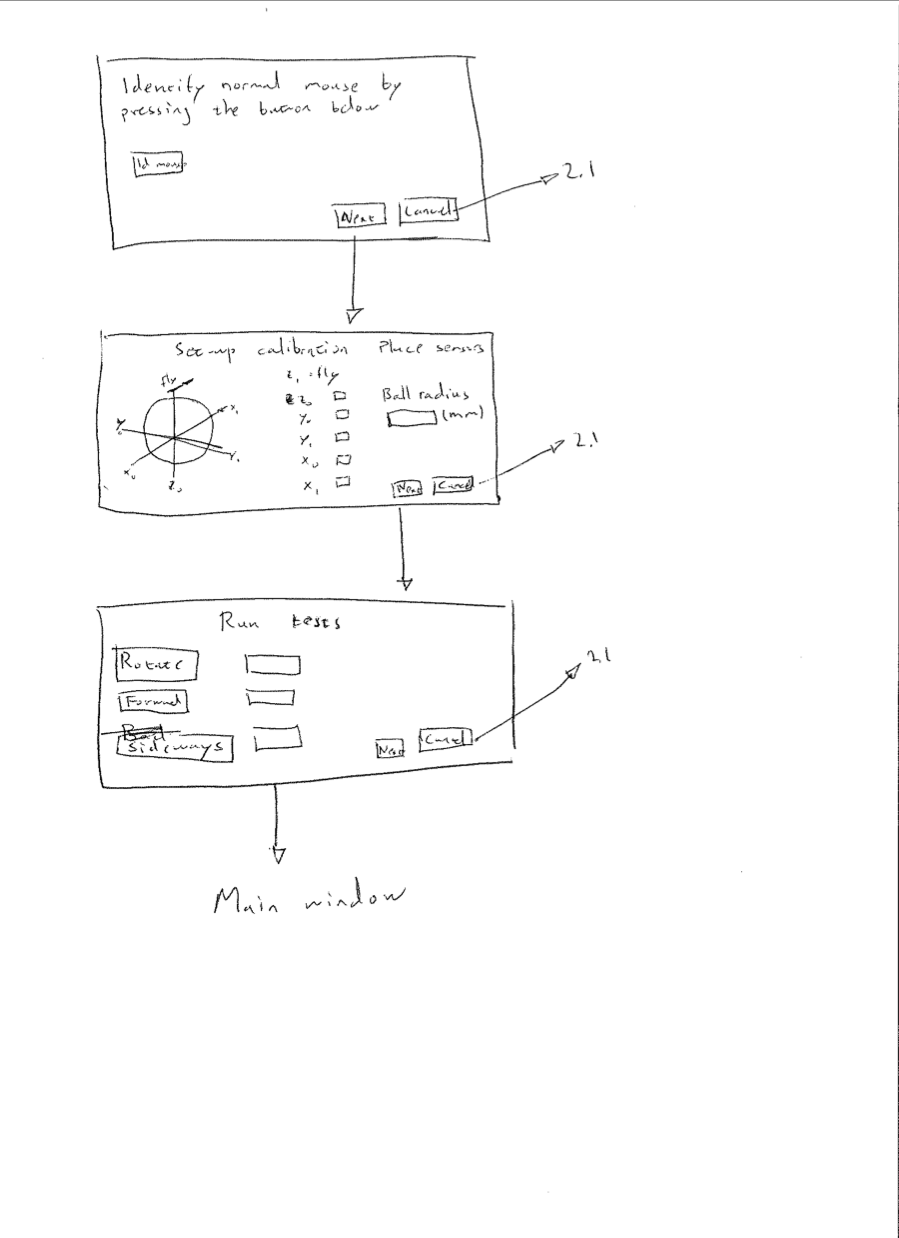


Figure 2 – Calibration wizard (2/2)

How to implement the interactive calibration part was the biggest issue when designing the GUI. The idea presented here is a step-by-step guide with clear descriptions for each of the steps. The relationship between the software and the hardware is not entirely intuitive and this makes understandable and clear instructions even more important than usual. After the calibration is completed the user is directed to the main window. The prototype is displayed here:

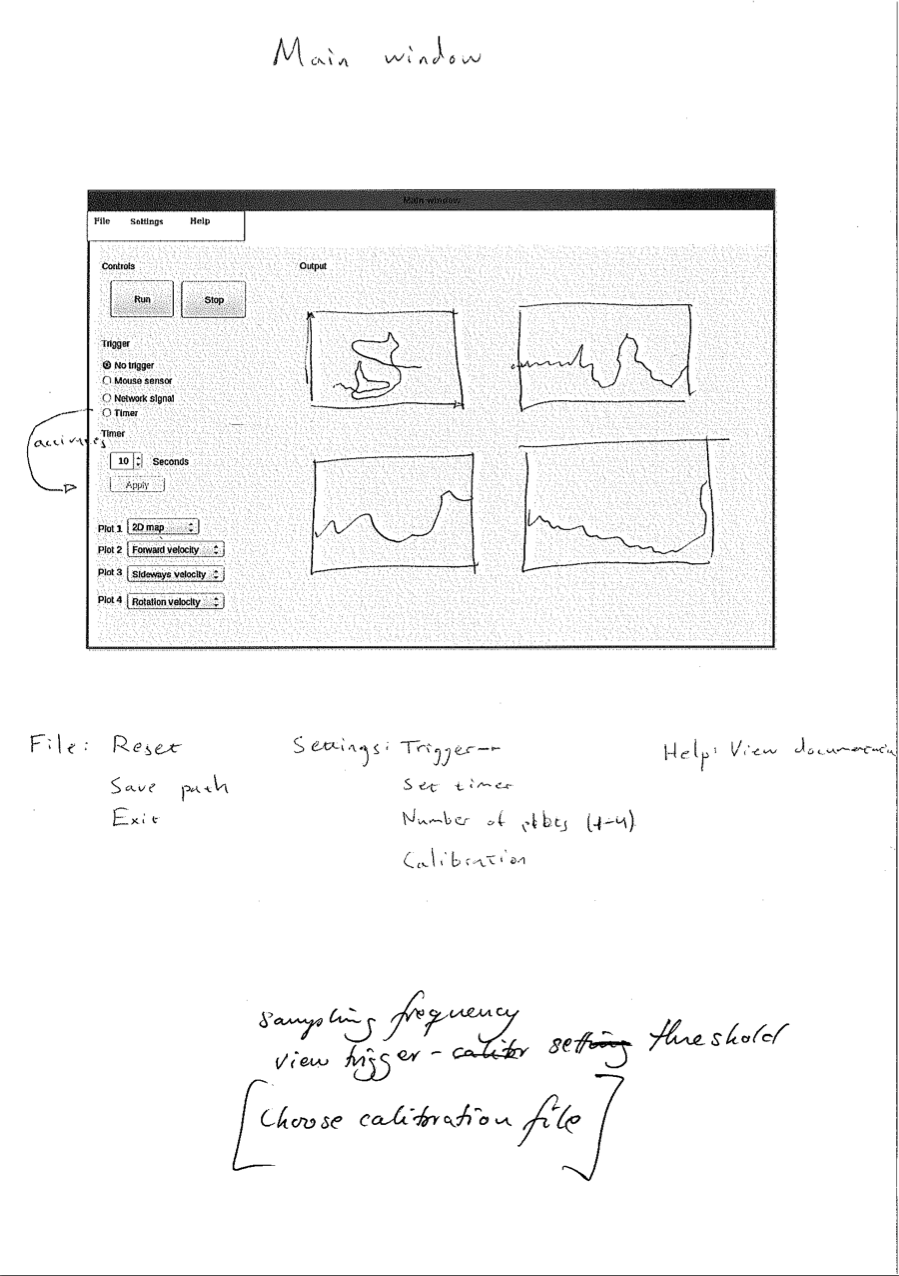


Figure 3 – The main window with controls for starting and stopping recordings and output of data

The main GUI is based on common design principles, all controls to the left and the output to the right makes it trivial for the user to locate necessary interaction items. The menu provides controls that are less important than the ones presented in the GUI.

1. Implementation

FlyTracker is implemented in Matlab and Python and the former provides the GUI and most of the calculations while the latter provides the core of the functionality. FlyTracker is also platform dependant and can only be run in Ubuntu Linux although changes to the Python module could make it more portable. In fact it would be fairly easy to replace the Python package right now as it is weakly coupled with the Matlab package.

* 1. The Pyhton module

There are three files in the Python package, DAQ.py, idMouse.py and utilities.py.

**6.1.1 DAQ.py**

This is the main file where most of the essential functionality is placed. It consists of the parent mouse class called AbstractMouse (though not technically an abstract class). IdMouse and SensorMouse both inherit from AbstractMouse which in turn inherits from Thread. SensorMouse provides functionality for reading raw mouse deltas and storing them in the global variable coordinates. IdMouse is used to identify which mice are the sensors and which mouse is the desktop mouse in the calibration step.

The MouseHandler class is also running in its own thread and is as the name suggest responsible for handling the mice. It initializes the mice and starts and pauses them accordingly.

As the data acquisition is run in Python but the user interaction is dealt with in Matlab there has to be a way to communicate between these two. The MouseHandler is never stopped unless a stop command is sent from the Matlab application (exception is if you run FlyTracker with a timer as a trigger). The command is sent via a named pipe (which requires Linux) and is stopping the listening thread. All mouse threads are set to daemon which means that when all non-daemon threads have exited the Python process will terminate. As the terminating threads will exit abruptly there is potential for minimal data loss (one or two data points).

As data is read it is also saved into a temporary txt file, the data is saved as a json-parsed dictionary object. This object holds two x-values and two-y values, one for each mouse sensor. The final value is a timestamp of that data point. This temporary data file will be opened and read by the Matlab package which will process and save it long term.

**6.1.2 idMouse.py**

IdMouse is used purely for identifying the mice .

**6.1.3 utilities.py**

Utilities.py provides some useful classes for saving and loading to file, logging exception and performing some necessary data processing.

Trigger

The triggering system is a major part of the functionality of FlyTracker. A trigger is basically an on-/off-switch for the recording. For example if the user uses a network trigger the recording won’t start as the user press “Run” but instead it will open up a network socket that listens for commands from a trigger client. When a client sends a “start” command the recording commences and runs until the client sends the “pause” or “quit” command or alternatively the user press stop.

The first triggering option is to use no trigger at all in which case you can manually control when FlyTracker is recording by pressing “Run” and “Stop”.

There is also an option to use a timer as a trigger which means that the user can set a timer for how long FlyTracker is supposed to record data.

* 1. The Matlab module

The Matlab module consists of two subpackages, model and view.

**6.2.1 Model**

The model holds the files that process the data, first by loading the temporary data from the txt-file that is created in the Python module.

Readdata.m is except for one of the GUI-files the most important part. As the name suggests, it is in this function that the temporary data file is read and parsed into the correct format. The readdata file also plot the current data continuously, updating the plot every hundred data points.

The raw data that is read is also transformed in the model package. The following two equations are used to calculate virtual translation movement and yaw movement in the fly’s coordinate plane.

Calculating the translation vector of length two, containing x- and y-movement:

*w\_m = beta\_.\*[cos(omega),-sin(omega);sin(omega),cos(omega)]\*[y2;y1];*

Calculating yaw rotation where *r* is the radius of the ball:

w\_mz = alpha\_\*t/(r);

*t* is given by:

*t = (x1+x2)/2;*

*if sign(x1) == -1 && sign(x2) == -1*

*t = max(x1,x2);*

*elseif sign(x1)\*sign(x2) == -1*

*if sign(x1) == -1*

*if abs(x1)<x2*

*if abs(x1)<t*

*t = x1;*

*end*

*else*

*if x2<abs(t)*

*t = x2;*

*end*

*end*

*elseif sign(x2) == -1*

*if abs(x2)<x1*

*if abs(x2)<t*

*t = x2;*

*end*

*else*

*if x1<abs(t)*

*t = x1;*

*end*

*end*

*end*

*elseif sign(x1) == 1 && sign(x2) == 1*

*t = min(x1,x2);*

*end*

The last algorithm process the horizontal component of each mouse sensor to reduce the yaw signal when the fly is not rotating. This algorithm works by the idea that when the values of the horizontal component of each sensor are very different yaw or even having opposite signs the signal should be reduced. As when the fly is trying to rotate these values cannot have opposite signs.

Saving is also done in readdata and is done in the end of each experiment so saving is not done continuously. However, the temporary data file holds all necessary information to recreate the data. So should FlyTracker exit unexpectedly or be closed before the recording has been stopped all data from that experiment can be recovered on FlyTracker start-up. The user will be promted to either recover the data or just delete the temporary data file.

**6.2.2 View**

All of the files holding functionality for the GUI and dealing with user interaction are placed in the view. The main window here is the main window of FlyTracker, the result can be seen in fig x and be compared by the initial prototype.

An important part of FlyTracker is the calibration,

**6.2.3 Data**

The data is saved into mat-files. Each file is initialized by pressing “Run” in FlyTracker and saved when the user presses “Stop”. Everything that is recorded in between is saved into the same data file. If you are using network trigger, recordings can also be paused and each pause then creates a new data block within the same mat-file.

Data is saved into .mat-files with the filename of the current time. Each data file consists of one 5 x n cell where n is the number of blocks. The elements of the first row in the cell consists of arrays of thrust delta position, second row is for sideslip delta position, third is for yaw delta position and the forth one is a time stamp. Each column represents data for that particular time stamp. The final row of the cell holds the starting time of that block, i.e. not an array but rather a date and a time.

* 1. Future expansion

A possible development of FlyTracker would make the entire experimental system to a closed loop system. With that means that as visual stimulus is presented to the fly, the behavior is recorded in FlyTracker and as a function of the behavior new visual stimulus is calculated. A closed loop experiment would set high demands of latency of FlyTracker and as the system is developed now with both a Matlab and Pyhon module it would be difficult to satisfy these demands. An alternative would be to possible migrating all data processing to the Python module and only display the result in Matlab. This would cut a lot of time as transfer of data between processes is currently the most time consuming part of FlyTracker. It would still be necessary to transfer the data to the Matlab module but as this would only be for displaying the output the time critical element would disappear.

Migration to Python would also mean better support for increasing the modularity and decreasing coupling. There is definitely work needed to be done in this area to make extensions, that are not even thought of right now, easier to implement.