Rovio Predator/Prey

# Planning

The tasks given present two goals. The first is to make a robot that can adequately traverse an arena filled with obstacles, and the second is to implement advanced techniques with exceptional functionality. After initial testing of the Rovio robots (see the ‘Testing’ section for further details), it was clear that a single system would not adequately meet both aims. In order to fulfil both of these aims, this document details two proposed designs. In order to create a well-programmed project, having these two separate goals is affected the planning of the program structure.

## Program structure

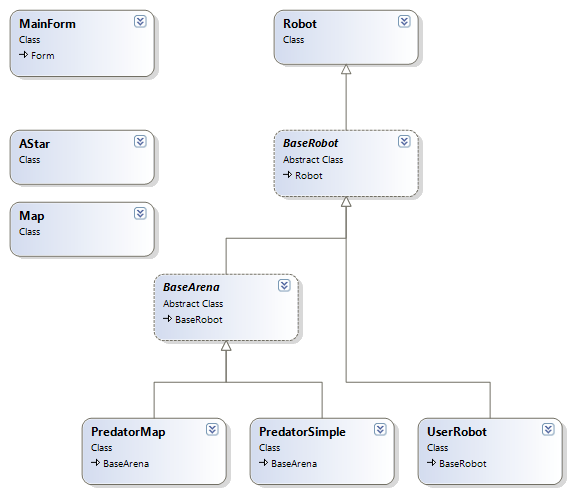
Due to the decision to create two separate implementations, there was careful deliberation regarding the good practice of reusing code. The program falls into three layers.

**Abstract ‘robot’ class**   
The topmost class. This holds functions to implement web API commands (such as receiving a camera image, and movement). This also contains very general image processing, such as colour segmentation, blob detection, conversion of image formats and the drawing of rectangles to a Bitmap. The methods for the implementation of keyboard input in derived classes also reside here.

**Abstract ‘arena’ class**  
Derived from the base robot class, this class provides functionality specific to the arena in use for this project. Variables for all prey, obstacles, and walls belong in this class, along with the necessary colour filters. Methods specific to this class include those for being able to find the faced direction in the arena, find the dimensions of encountered objects, and perform further analysis of images. This class calls the image processing classes found in its base class, and performs specific analysis of the collected data within itself.

**Derived classes**  
Classes can derive from either of the aforementioned classes. The implemented user controls class is derived from the base robot class because it does take into account any specific occurrences found in its environment: all it wants to do is look and move based on the keyboard functionality (from which is taken in by overriding the base class’s keyboard method). A predator or prey class derives from an arena class, and can thusly use all collected data regarding happenings in the environment.

The application would have been equally functional regardless of whether the arena class was separate from the base class. A design decision was made to separate the functionalities found within them both so that should the project be revisited with a new environment, the arena class could be fully replaced to suit the new territory without having to delve into base robot class within which all functionality is necessary for a functioning robot. Whether the application implements predator or prey remains undisclosed to emphasise that the design decisions allow the implementation of either to any scale simply by creating a new derived class. The implemented derived classes are two predator classes, one with simple functionality and one with advanced mapping techniques.

Figure 1 – Class diagram displaying the considered structure of the application.

## Finite State Machine

The ‘PredatorSimple’ class implements a finite state machine approach for finding the prey. It consists of four states. The tactics shown for this predator display an aggressive searching approach. In order to minimise wasted movement, the predator patrols around obstacles until the prey is found, at which point it directly approaches the prey.

**Search for obstacle**  
The prey incrementally rotates until an obstacle is within its vision.

**Move around obstacle**The predator moves around the block. This state triggers a series of movements, detailed in the implementation section.

**Approaching** – Move towards the prey, making use of a bang-bang controller to account for movement error and to keep the target in view.

**Search for recently lost prey** – This state triggers for a short amount of time after sight of the prey is lost. The predator rotates slightly in the direction that the prey went out of sight. If the prey is seen again the ‘Approach’ state is entered. Otherwise, the cycle restarts at the ‘Search for obstacle’ state.

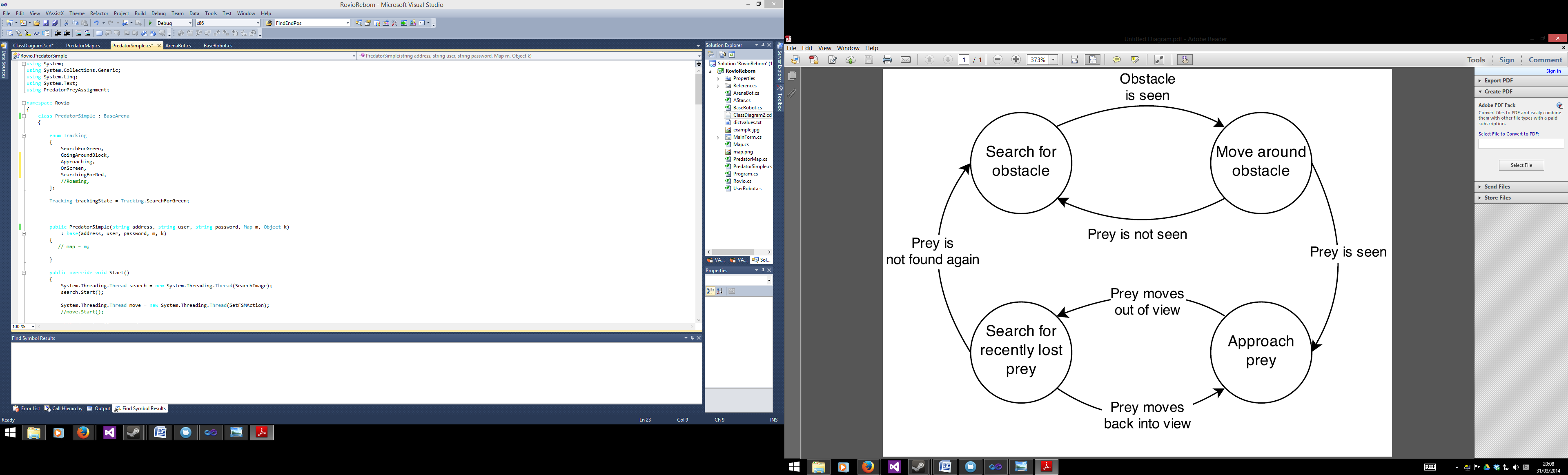
****

Figure – Finite State Machine for the simple implementation of predator

Blind spots are not initially clear, but by circling around obstacles continuously, areas both in initial view and obscured by initial view become equally considered over time and constant movement makes prey evasion considerably more difficult than an approach that uses stationary scanning.

The implementation of the finite state machine makes use of enumerations, which are values types whose values are limited to a defined number of symbolic names (Sharp, 2010, 173). For PredatorSimple, the derived class creates the ‘Movement’ thread, which handles the state machine actions. The method called by the state machine thread checks the variables, the data for which is collected in the Arena class, and uses the values to decide which state the Rovio should be in. The initial state is ‘Search for obstacle’.

If the prey enters view during any phase, approaching they prey becomes the highest priority and the application enters the ‘approaching’ state. The application performs a check at the beginning of every loop to see if the prey has entered view, and will immediately switch states accordingly.

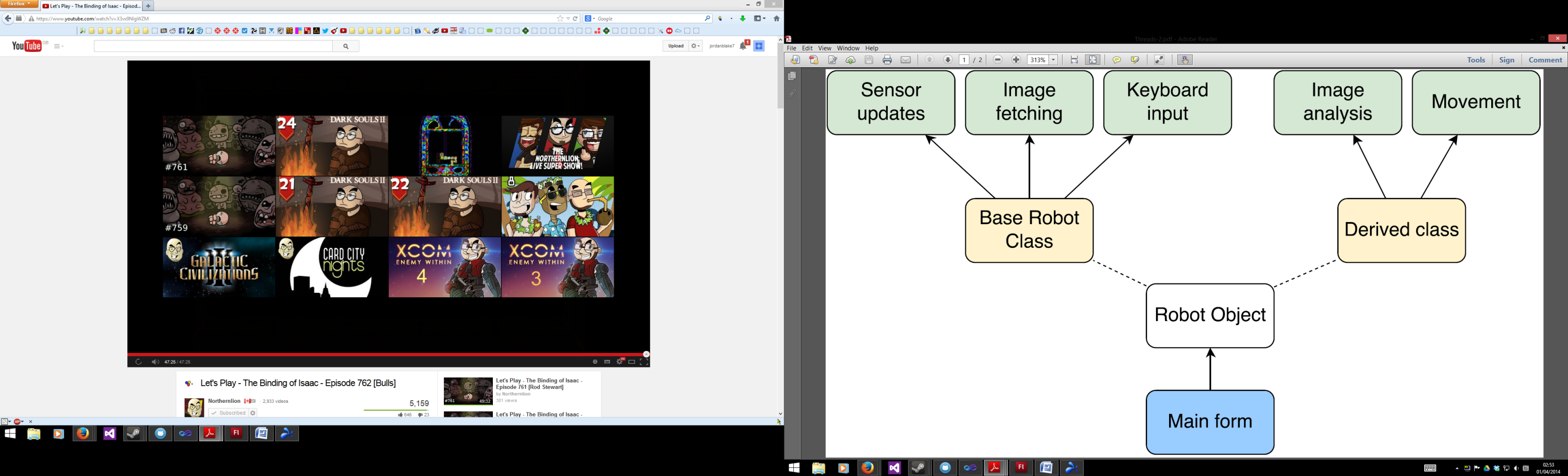
# Implementation

## Threading

With the implementation of threading, this application performs actions alongside each other. For example, it is possible to segment an image whilst the Rovio executes a movement command. The Rovio holds a severe limitation, allowing no concurrent operations (i.e. only allowing the execution of a single command at a time). An attempt to send concurrent input/output operations results in an application-breaking error from the Rovio. This would not be an issue on a single thread as everything a one thread sequentially executes commands, but multiple threads introduce the (very likely) possibility that the Rovio receives simultaneous commands. To counter this issue, the application makes heavy use of the **lock** keyword. The lock keyword takes an object as an argument. When a thread enters a lock region, the object passed indicates to other threads that it is in use. If other threads reach a lock region whilst the object is in use, they will ‘queue’ and wait for the lock object to become free again. Assurance is given that the Rovio will only receive one command at a time by placing all API calls within their own lock regions. Lock regions are included in all abstract class functions that make an API call so that derived classes can safely make use of API commands without worry.

The main form creates a thread, which runs the base robot. From the base robot class, two threads run: one for receiving the camera image, and one for accepting keyboard input from the main form. The derived classes execute their own threads for searching through the camera image and for their own movement methods. Allowing the base thread to receive camera images as quickly as it can fetch them ensures the Rovio is always using the very latest data from the environment, increasing the probability that its assumptions are correct. At the beginning of the segmentation thread loop, there is a check to see if the latest image matches the last segmented image. On the occasion that the application has not fetched a new image during the course of segmenting the previous image (which can happen due to latency issues), there is no unnecessary performance wastage by repeatedly processing the same image. The result is that the performance of image capturing and segmentation is extremely quick.

Figure - A visual representation of the threading used in the application.

Another considerable danger with threading is that, if not handled correctly, threads will not terminate safely. C#’s threading contains an ‘abort’ function, but it is not advisable to use this as threads may not run fully and an exception may be problematic within the program (MSDN, 2014). Due to how the main form handles switching between derived class types (reinitialising an object of the base class as the chosen derived class), it has to be a certainty that threads are fully terminated before re-initialisation of the object occurs. The chosen solution is to use a single Boolean variable as the condition for all thread loops. When the robot object is to be altered, this Boolean can be changed and the thread ‘Join’ function can be used to wait for all threads to naturally expire before continuing with the application. Due to the nature in which the ‘abort’ method ends threads, letting threads naturally run their course is a much safer option than abruptly calling for their termination.

## Localisation

The ability to map an environment is an advantageous feature for a robotic system to have. However, the map is only as reliable as the input received by the robot, which make Rovios difficult to deal with for this purpose. With the lack of reliable odometry, no dedicated sensors for measuring distances, or even a reliable way to deduce distance travelled (see the testing section for further information), the camera is the only way for the Rovio to navigate its environment.

As the class structure allows for image processing dedicated to an arena, the chosen method for mapping is to use a hard coded map rather than attempt to make the Rovio map walls accurately. The scale used for the map is one pixel per centimetre. Creating cells at this scale is inadvisable since the resulting map would be 78,000 cells large (with a measurement of 260cmx300cm, including inaccessible space), so a division of ten is made to create a 26x30 cell grid, resulting in the considerable smaller grid size of 780. As everything the Rovio expects to encounter is greater than a single cell, division of ten is adequate. A division of twenty would also divide well and a single cell would be greater than the smallest object in the arena (another Rovio, at 25x27cm) but the added accuracy of a 10cm-per-cell map weighs in its favour.

Two variables are required for this mapping method: a heading relative to a position defined as north, and a distance from the faced wall. Matrix mathematics permits the ability to rotate in the heading direction and translate backwards from the wall the robot is facing. The position found from this method is as accurate as the data received from the sensor. The only sensor for this purpose is the vision sensor, which picks up a considerable amount of noise. Getting the distance from the wall is possible by measuring the thickness of the blue line along the arena walls, which is equally thick along the entire arena boundary. By measuring the thickness of the blue line in pixels from a distance of 1m, it is possible to estimate the distance from the blue line with a reasonable degree of accuracy.

Getting an accurate measurement of the heading is more complicated because of the amount of factors involved. The arena walls are uniquely marked with a blue line at different heights. By finding the colour of the faced wall, the number of segmented colours (to see whether the blue line lies along the edge or the centre of the wall) and which colour wall is on either side of the screen (in the case of a corner), the faced direction can be found. Facing corners provides the most accurate result as two segmented rectangles at once give more reliability than one, since a single rectangle from image processing is much more subject to external noise and not comparable against expected values of other data. For example, the white walls are the most difficult to accurately analyse because the segmented rectangles often extend beyond the boundary of the wall. When viewing a single white wall (the most unreliable colour in the arena to segment), the resulting rectangle data can only be assumed as correct. When viewing a corner and observing the more reliable yellow wall alongside the white wall, the rectangle results are checked against each other to validate their accuracy.

To calculate the heading, XNA’s vector mathematics library is used. Fifty equally spaced reference points are stored (one for each degree of the Rovio’s 50° field of vision), and how many reference points lie on each wall are counted. A cumulative vector stores a vector for each reference point, which is relative to north based on which wall the reference point lies. At an angle of 65° (assuming north as 0°), one third of the fifty reference points will lie on the north wall and the remaining two thirds will lie on the east wall. The vector stored for a north reference point will be 0, -1 (as north is –Y and south is +Y) and the east vector will be -1, 0 (a from an east facing position, north lies at -1 along the X axis). The vectors are rotated depending on where they lie in the field of view (from -25 to +25) all the vectors are cumulatively added. The resulting vector is then normalised, and the inverse tangent of the normalised vector provides the heading in radians.

Noise has a large effect on the result of the wall distance and heading angle. Use of linear interpolation reduces the error for both of these values. The final position is also updated with use of linear interpolation.

## Bayes Filter and A\* Pathfinding

Bayesian filtering is a method of probabilistic estimation, comparing a newly calculated probability to the last calculated probability, and weighing those values against the accuracy of the map and sensors. Bayes filtering initialises values to 0.5 (on a scale of 0 to 1, making the initial value equally probable and improbable.

This implementation uses a two dimensional Boolean array indicate whether a space is occupied or not. The maps indicate occupation when the probability of a cell passes a high threshold. There are two 2D double arrays holding the probability for each type of sensor – prey and obstacle. Evaluating these probabilities separately is necessary to act accordingly dependent on what the robot encounters. Using multiple maps is also justification for the decision of scaling the map size.

Probability can only change for encountered objects directly within the robot’s vision, so a viewing cone shows the reach of the field of vision. Testing showed a distance of 1.5m for the viewing cone is best because at a further distance the red block used to indicate a Rovio appears small enough to be comparable to background noise, making readings past this point unreliable. As previously mentioned, the angle of vision measured for the Rovio is 50°.

Pathfinding is implemented using the A\* algorithm, chosen over Djikstra’s algorithm for its increased efficiency, which is important given that the path is subject to frequent change. The decision of destination is the highest probability cell on the ‘prey’ Bayes map, past the aforementioned threshold. Addition to the algorithm’s closed list is with cells displaying a probability passing a threshold on the ‘obstacle’ Bayes map.

When there is detection of an obstacle, the system knows that the Rovio’s path is obscured. This can present problems for probabilistic estimation if a destination is set based on the detected position of the prey and then temporarily obscured (since cells behind the obstacle appear ‘in view’ and probability decreases regardless of their content). To compensate for this, alteration of the viewing cone happens when an obstacle is in view. Points of the viewing cone are added at points on an obstacle, as seen in figure X. The viewing cone is stored as a points array and only cells lying within the viewing cone are checked using a method to check if a point lies in an array of points (Windows Dev Center, 2007).

# Testing

Accuracy of measurements taken from the Rovio are evaluated so that their reliability can be measured for use of Bayes filtering.

Wall distance accuracy.

|  |  |  |
| --- | --- | --- |
| **Distance from wall** | **Average distance measured** | **Standard deviation** |
| 20m |  |  |
| 50cm |  |  |
| 100cm |  |  |
| 200cm |  |  |
| 300cm |  |  |

Average time per thread loop across a one-minute execution (excluding movement and keyboard input).

|  |  |  |  |
| --- | --- | --- | --- |
| **Thread** | **Average loop time** | **Number of times called** |  |
| Receiving image |  |  |  |
| Receiving sensor information |  |  |  |
| Image analysis |  |  |  |

# Reference list

MSDN (2014) *Thread.Abort Method*. [online] Washington: Microsoft. Available from <http://msdn.microsoft.com/en-us/library/ty8d3wta.aspx> [Accessed 13 March 2014].

Sharp, J. (2010) *Microsoft Visual C# 2010 Step by Step*. Washington: Microsoft Press.

Windows Dev Centre (2007) *Determine if the point is in the polygon, C#* [online] Washington: Microsoft. Available from <http://social.msdn.microsoft.com/Forums/windows/en-US/95055cdc-60f8-4c22-8270-ab5f9870270a/determine-if-the-point-is-in-the-polygon-c?forum=winforms> [Accessed 22 March 2014].