Rovio Predator

# Planning

There are two different implemented robot types: one which makes use of a finite state machine, and another which uses advanced techniques to map the arena. The decision to create two implementations came from the hypothesis that mapping features would not work well with the specifications provided by the Rovio.

## 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. Functions include:

* Web API commands (e.g. receiving a camera image, movement).
* General image processing (colour segmentation, blob detection, conversion of image formats and the drawing of rectangles to a Bitmap)
* Abstract keyboard input function to be implemented by derived class.

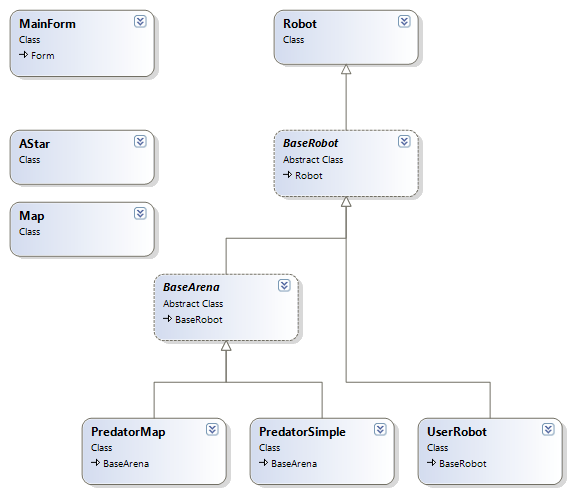
**Abstract ‘arena’ class**  
Derived from the base robot class, this class provides functionality specific to the arena in use for this project. Variables for all prey, obstacles, and walls belong in this class, along with the necessary colour filters. Methods specific to this class include:

* Finding direction faced in the arena.
* Getting dimensions of encountered objects.
* Further image analysis.

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. A class derived from the Robot class will be for non-specific uses (i.e. user input). Classes derived from the ‘arena’ class will have functionality specific to the arena environment.

Separating the robot class and the robot class gives no extra functionality. The design decision to separate them was made so that the arena class could be replaced to segment images in a new environment without having to touch basic functionalities in the base robot class, making the project as versatile as possible.

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**  
  Prey incrementally rotates until an obstacle is within its vision.
* **Move around obstacle**Predator moves around the green.
* **Approaching**

Move towards the prey, using a bang-bang controller to account for error.

* **Search for recently lost prey**

Triggers for a short amount of time after prey is lost. Predator rotates slightly in the direction that the prey went out of sight. If the prey is not found, ‘search for obstacle’ begins again.

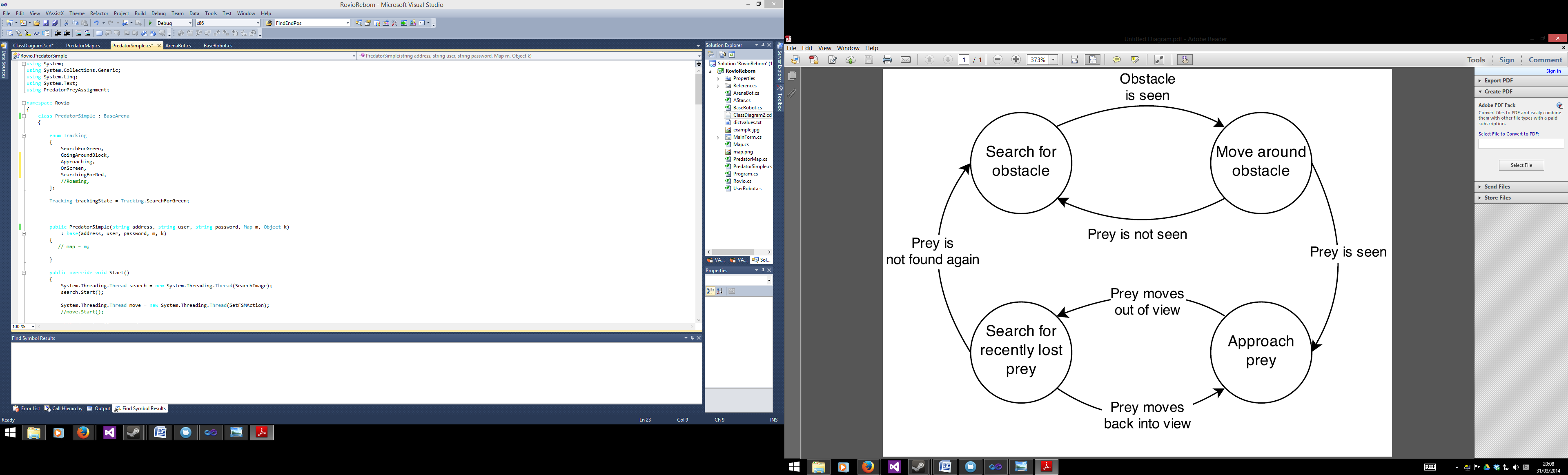
****

Figure 2 – Finite State Machine for the simple implementation of predator

Blind spots are not initially clear, but circling around obstacles continuously equally considers areas initially in view and obscured by moving into the non-visible space. Constant movement makes prey evasion considerably more difficult than an approach that uses stationary scanning.

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 actions. This thread checks the variables, the data for which is collected in the Arena class’s image processing, 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 is in view..

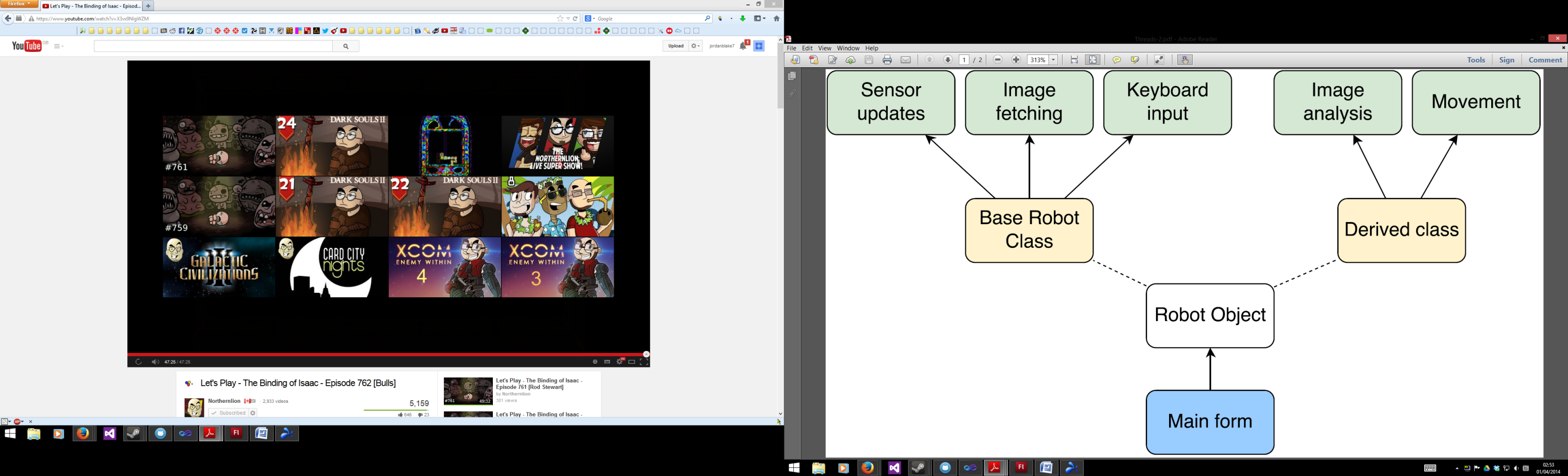
# Implementation

## Threading

The application runs multiple threads concurrently. 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, the application makes heavy use of the **lock** keyword. ‘Lock’ takes an object as an argument. When a thread enters a lock region, the object passed indicates 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. This assures the Rovio will only receive one command at a time by placing all API calls within lock regions.

The main form creates a thread for the robot object. From the base class, two threads run: one for receiving the camera image, and one for accepting keyboard input. The derived class executes its own threads for image processing and movement. At the beginning of the image processing 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.

Figure 3 - A visual representation of the threading used in the application. The Robot Object is an instance of the derived class. The base class begins some threads and the derived class always begins the two linked, but can create more if necessary.

Another danger with threading is that, if not handled correctly, threads will not terminate safely. C# contains an ‘abort’ threading function, but using this may prevent threads from running fully, creating potential exceptions (MSDN, 2014). To make sure threads are executed fully before the application reinitialises a robot object, a single Boolean variable is used as the condition for all thread loops. Changing this variable to false makes loops exit when they end. The ‘Join’ function is used to wait for all threads to naturally expire before continuing with the application, resulting in a safe termination.

## Localisation

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 track its environment.

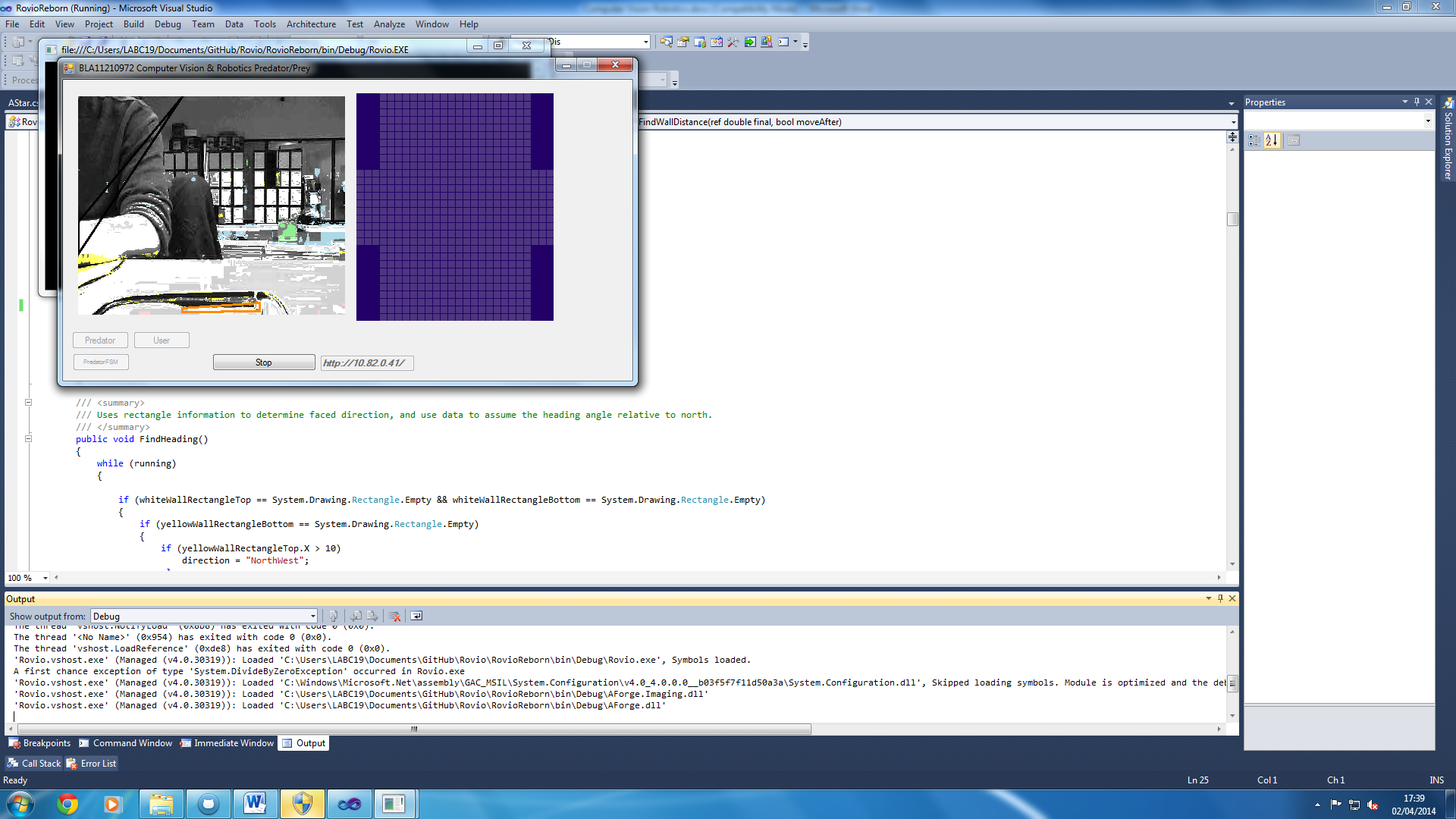
A hard coded map is used rather than attempt to make the Rovio map walls accurately. Creating cells at a scale of one pixel per centimetre is inadvisable since the resulting map would be 78,000 cells large (with a measurement of 260cmx300cm, including inaccessible space), so a division of ten is made 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 suitable for keeping a good degree of accuracy.

Figure 4 – The map divided into cells, and overlaid with a red and a blue probability map, with the alpha value representing the probability of detected objects.

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 allows rotation in the heading direction and translation 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.

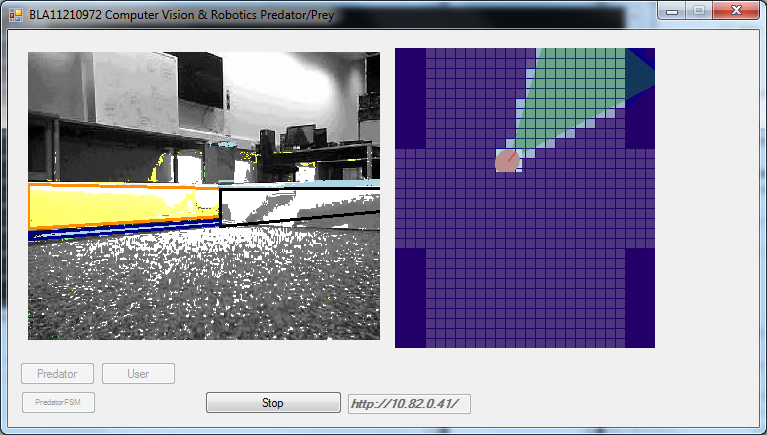
Getting an accurate measurement of the heading is more complicated because of the amount of factors involved. The arena walls are uniquely marked with a blue line at different heights. By finding the colour of the faced wall, the number of segmented colours (to see whether the blue line lies along the 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.

Figure 5 - Localisation facing the northeast corner.

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 (Riemer’s, 2011).

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.

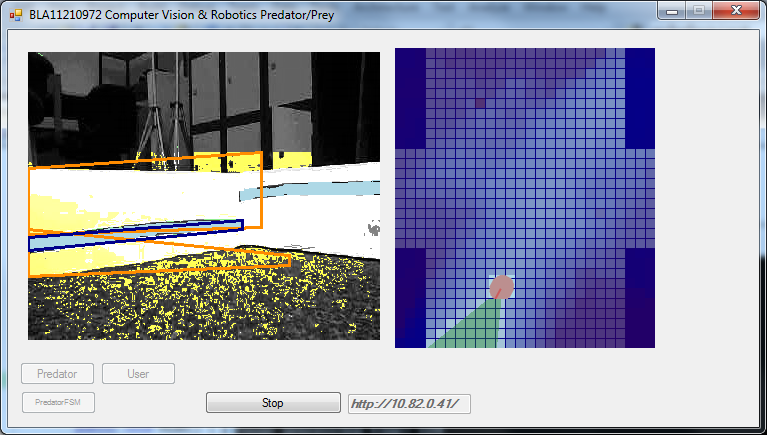
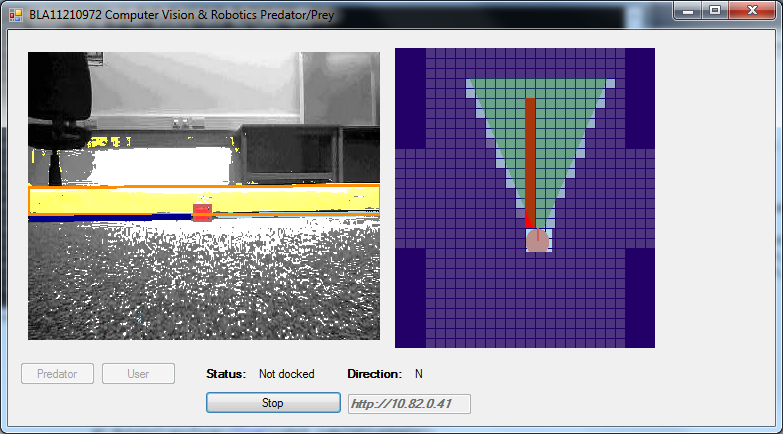
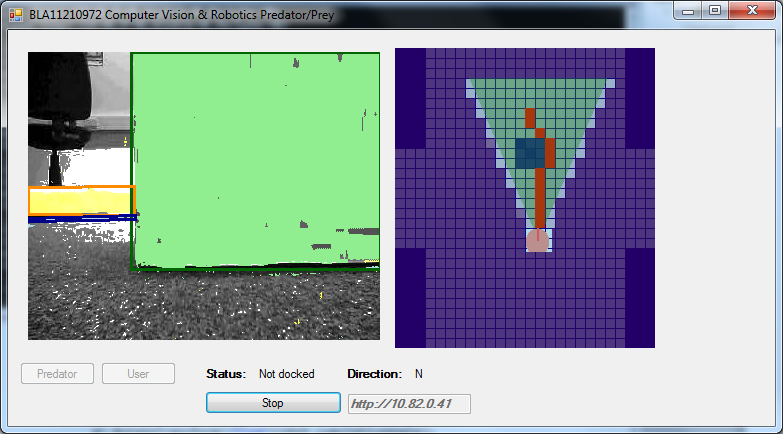
Probability can only change for encountered objects directly within the robot’s vision, so a viewing cone shows the reach of the field of vision. Testing showed a distance of 1.5m for the viewing cone is best because at a further distance the red block used to indicate a Rovio appears small enough to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

Figure 6 - Probabilistic estimation after movement.

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).

Figure 7 – A\* pathfinding with and without an obstacle in path (without checks to decrease probabilities not in view.

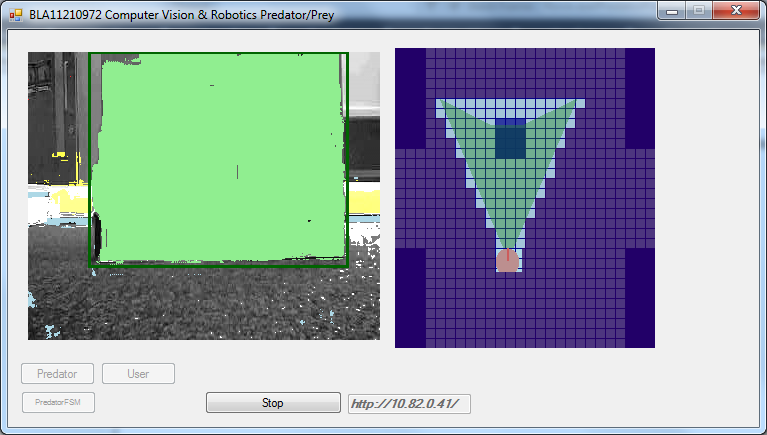


Figure 8 - Altering the viewing cone to remove checks for obscured parts of the field of view.

# 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 from ten readings

|  |  |  |
| --- | --- | --- |
| **Distance from wall** | **Average distance measured** | **Standard deviation** |
| 20cm | 0.3443 | 0.047495 |
| 50cm | 0.5926 | 0.002503 |
| 100cm | 1.2157 | 0.048155 |
| 150cm | 1.7964 | 0.115749 |
| 200cm | 2.301 | 0.101045 |

Measurements at various differences were taken to judge the accuracy and reliability of results. Interestingly, the closest distance was not the most accurate result. A probable reason for a more reliable result at 50cm rather than closer is because when the Rovio is very close to a wall, it can cast a shadow which affects the readings it takes from the image. This was not corrected by altering the lighting as the best test results are those that most accurately reflect real world occurrences.

## Resolution testing

|  |  |  |
| --- | --- | --- |
| **Resolution** | **Average loop time** | **Standard deviation** |
| 352x288 (CIF) | 62.8 | 56.99278707 |
| 640x480 (VGA) | 540.5 | 491.5079405 |

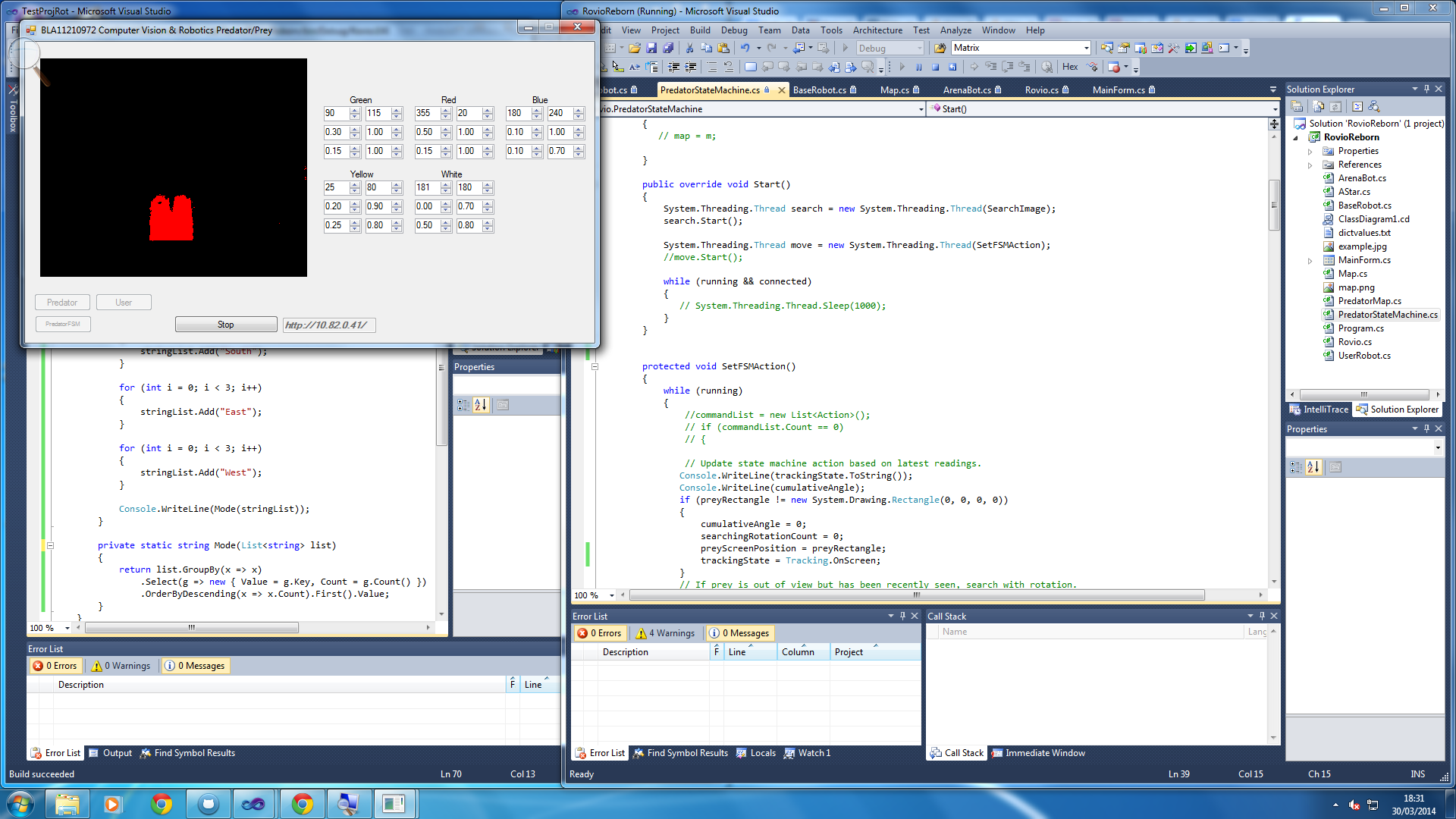
A higher camera resolution is desirable for higher quality results from image processing, but it was hypothesised that a higher resolution image would take longer to take from the camera. The two highest resolution camera settings were used for testing this. Overall it was shown that a the VGA camera image took significantly longer to receive than the lower resolution CIF image, and with a less stable result. CIF image quality is adequate for image processing, so it was selected over VGA for the faster capture time.

## Average time per thread image loop across ten calls

|  |  |  |
| --- | --- | --- |
| **Thread** | **Average loop time** | **Standard deviation** |
| Image processing (1) | 325.6 | 111.5409242 |
| Image processing (2) | 559.4 | 247.6606998 |

Two image processing results were obtained: the former is as the only Rovio on the network, and the latter at a time when the network was busy. Observation shows that image processing loop takes a longer time to run through when the network is busy rather than when it is free. Although the calculation time is machine-dependent, objects used in the image processing loop are within lock regions for safety. With a larger amount of latency, other threads will take a longer amount of time, leading to a longer image processing loop. The standard deviation for the higher latency result reflects that the processing time is more unstable.

## Image Segmentation

Values for segmenting images were performed qualitatively using form controls to alter values and see the results instantly. By performing changes to the image analysis whilst the Rovio is running and trying to find the prey, the values can be altered to work as well as possible with the different lighting conditions across the arena.

# Evaluation

A large proportion of the limitations in this project come from the robot in use, and the shared network for testing used by many people at a time. Network latency has been shown to be a large factor in performance decreases. Due to the shared testing environment, anomalies were prevalent. It is entirely possible that some of the results (e.g. the resolution test) are unreliable due to latency, but it is not possible to explore this in the current setting. A stable testing environment could allow for results that are more thorough.

The implementation could have been improved by making use of probabilistic estimation and pathfinding in a functional robot. The ideal design is to gather clusters of the areas with the highest probability (initially 0.5, as unknown areas) and move to them. Placing pathfinding destinations directly behind obstacles would have also worked well. Unfortunately many problems arose in the development process relating to accurately localising the Rovio and moving it to data gathered from the map, which is why that implementation is not fully functional. The concepts held within the program show promise, but not result. The initial hypothesis of a simple implementation providing the best result for a robot with the Rovio’s specifications was correct.

# Reference list

Grootjans, R. (2011) *Direction to Angle*. [online] Antwerp: Riemer’s. Available from <http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2D/Direction_to_Angle.php> [Accessed 18 March 2014].

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].