For Office Use Only

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 features the Rovio provides.

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.

For Office Use Only

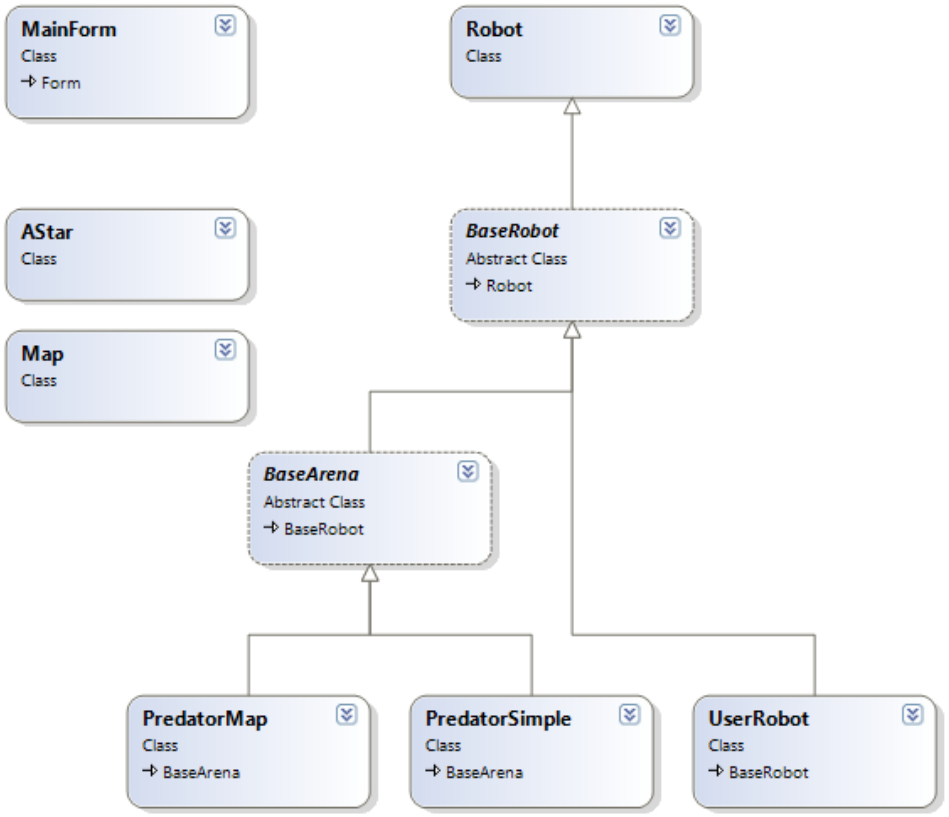

Figure 1 – Class diagram displaying the considered structure of the application.

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 obstacle.
- **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.

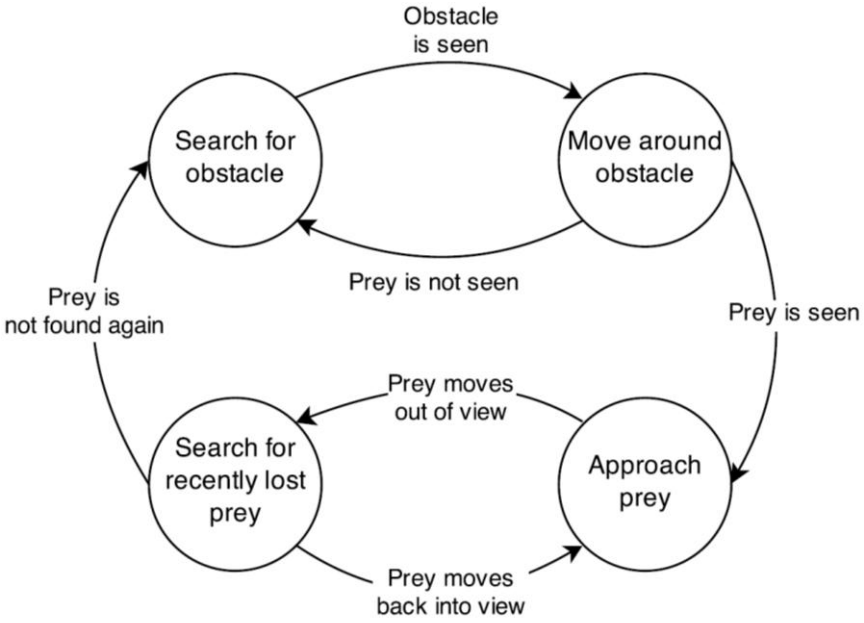

Figure 2 – Finite State Machine for the simple implementation of predator

Blind spots are not initially clear, but circling around obstacles equally considers areas initially in view, and those originally obscured by moving into out-of-view 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 value types whose values are limited to a defined number of symbolic names (Sharp, 2010, p. 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 them to decide which the current state. The initial state is 'Search for obstacle'.

If the prey enters view during any phase, approaching becomes the highest priority and the application enters the relevant 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 API. This would not be an issue on a single thread as every command would sequentially execute, 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 robot 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.

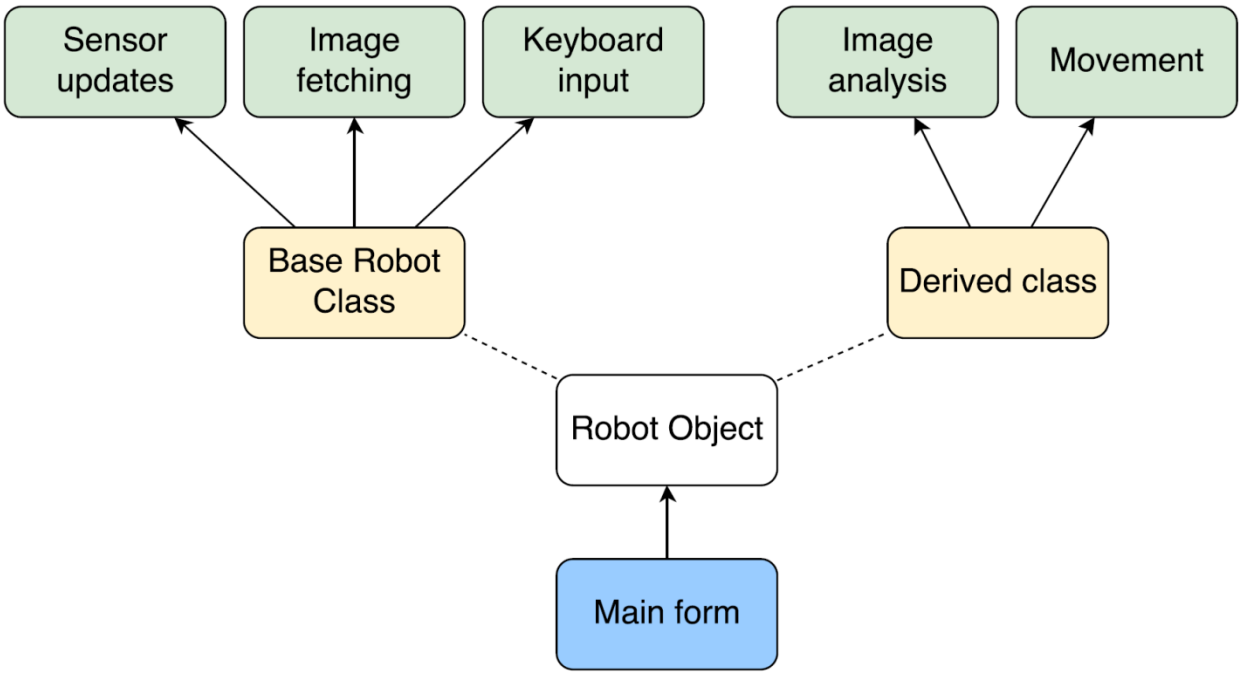

Figure 3 - A visual representation of the threading used in the application. The Robot object is an instance of the derived class. The base class begins some threads and the derived class always begins the two linked, but can create more if necessary.

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, the camera is the only way for the Rovio to track its environment.

The choice of a hard coded map over mapping the walls as the Rovio moved was chosen to get the most accurate result for this specific arena. 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.

Two readings are required for this mapping method: the heading for the direction faced, and the distance from the faced wall. North is 0°, defined as the centre of the yellow wall with the blue line at the bottom. The heading angle is used to find the point around the perimeter of the map that is being faced, and then the position can be found by translating backwards from that point by the distance from the wall and at the rotation provided heading (using matrix mathematics, commonly used in game programming). The distance from the wall is found by measuring the thickness of the blue line in pixels at the distance of 1m and scaling the recorded distance based on that value. The same method is used to find the distance of the prey and obstacles when they are on screen.

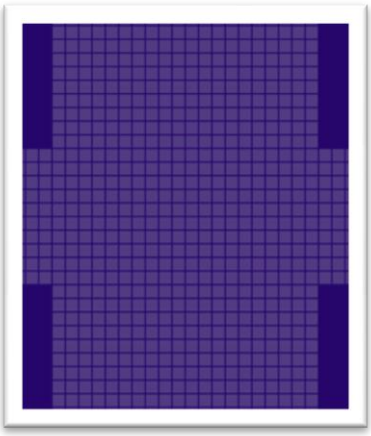

Figure 4 – The map divided into cells, and overlaid with a red and a blue probability map, with the alpha value representing the probability of detected objects.

The arena walls are uniquely marked with differing blue line 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. A corner will provide the most accurate result as comparing two segmented results against each other is more reliable than accepting the value of one.

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 sums vectors for each reference point, which is relative to north based on which wall the reference point lies. At an angle of 65° 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 (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). The resulting vector is normalised, and the inverse tangent of the normalised vector provides the heading in radians (Grootjans, 2011).

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

Bayesian filtering and 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. Initial values are 0.5 (on a scale of 0 to 1), making the cell value equally probable and improbable.

This implementation uses a two dimensional Boolean array indicate whether a space is occupied. The map indicates occupation when the probability of a cell passes a high threshold. There are two 2D double arrays holding the probability for each type of potential encounter – 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, since larger maps would lower performance.

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.

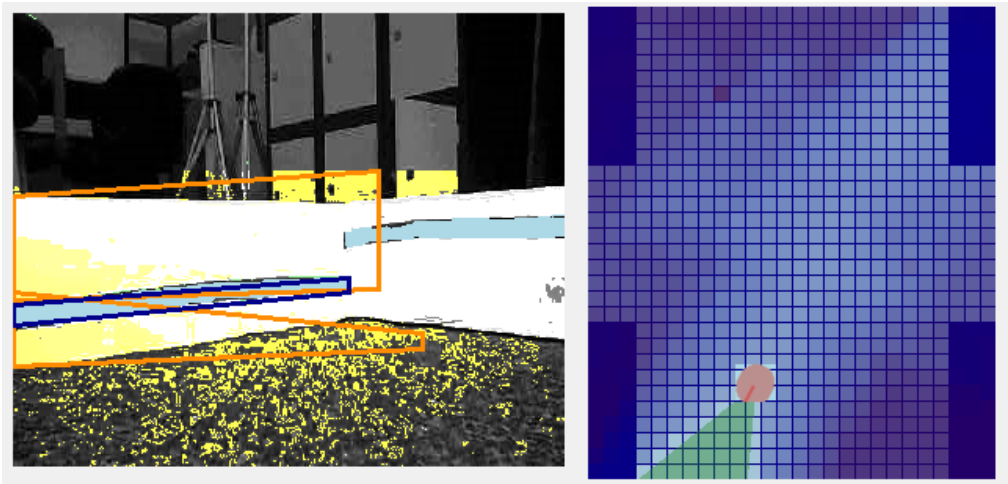

Figure 5 - Probabilistic estimation after movement.

Pathfinding is implemented using the A* algorithm, chosen over Dijkstra's algorithm for its quicker performance, which is important given that the path is subject to frequent change. The destination is the highest probability cell on the 'prey' map, past the aforementioned threshold. Cells with a high probability on the 'obstacle' map are added to the initial closed list of A*.

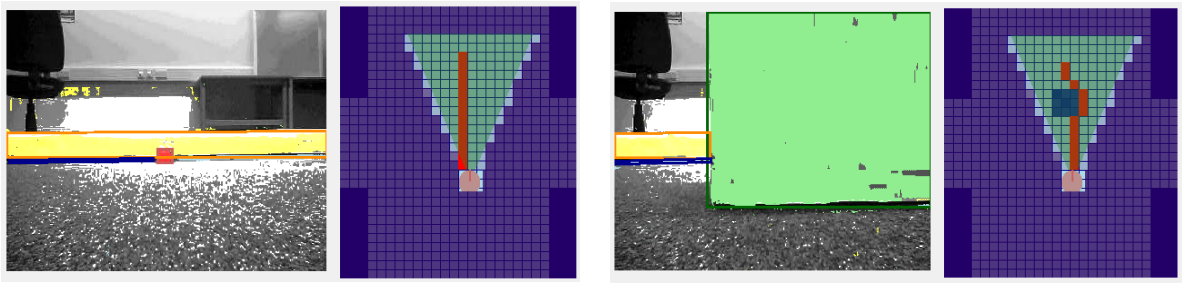

Figure 6 – A* pathfinding with and without an obstacle in path (without checks to decrease probabilities not in view.

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. The corners of the obstacle are added to the viewing cone, as seen in figure 7. The viewing cone is stored as a System.Drawing.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).

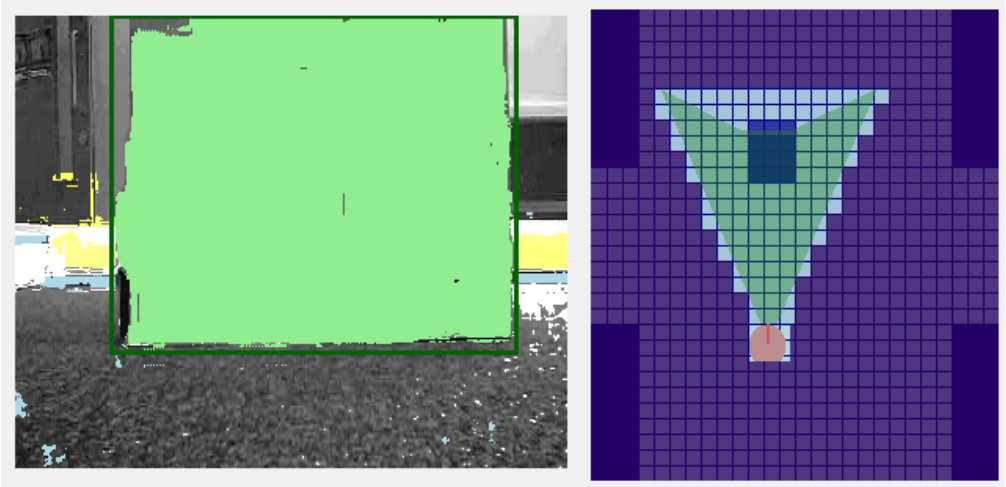

Figure 7 - Altering the viewing cone to remove checks for obscured parts of the field of view.

Testing

Wall distance accuracy from ten readings

Distance from wall	Average distance measured (m)	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 possible reason for a more reliable result at 50cm rather than closer is that when the Rovio is very close to a wall, it can cast a shadow that 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 (ms)	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 receive from the camera. The two highest resolution camera settings were used for testing. Overall, the VGA camera image took significantly longer to receive than the lower resolution CIF image, and with a less stable result time. CIF image quality is adequate for image processing, so the faster capture time resulted in its selection.

Image processing time across ten calls

Thread	Average loop time (ms)	Standard deviation
As only robot on the network	325.6	111.5409242
Sharing network with other robots	559.4	247.6606998

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 and keep image processing objects queuing, leading to a longer image loop. The standard deviation for the higher latency result reflects that the processing time is more unstable when sharing the network.

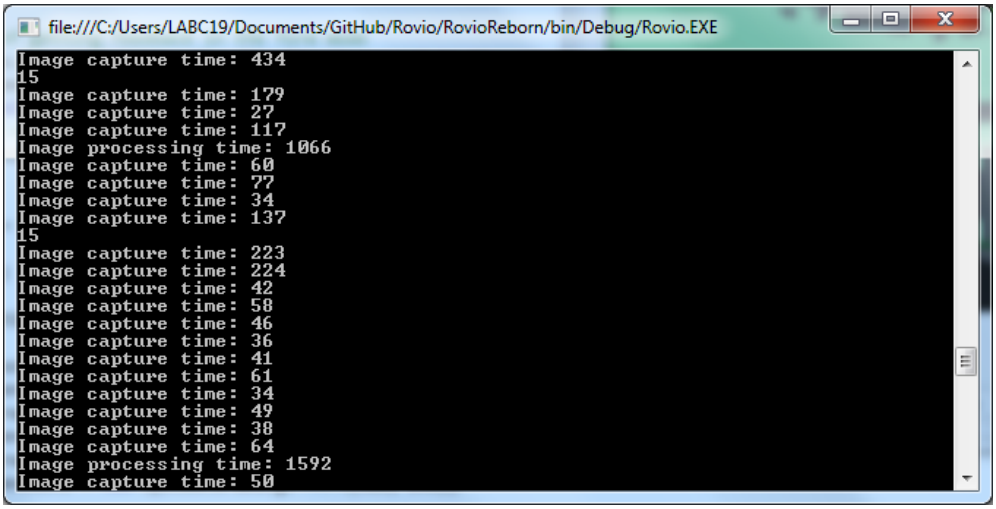

Figure 8 – Black box testing using stopwatches pre-processor directives.

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 were altered to work as well as possible with the different lighting conditions across the arena. HSL filtering (hue, saturation, and luminance) was chosen over RGB colour filtering, as the latter is prone to extreme variance in results with changes in lighting condition. HSL colour filtering proved to be more versatile in tests where lighting was drastically changed (in dark areas, or lit with a torch). The segmented arena objects were:

- Prey (red)
- Obstacle (green)
- North/south walls (yellow)
- West/east walls (white)
- Wall line (blue)

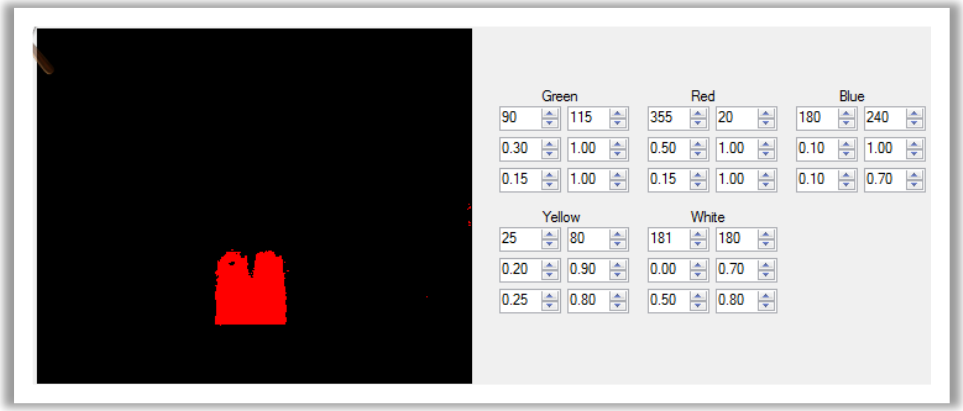

Figure 9 - Example of a segmented image output. Rectangles can be found using the AForge Blob Counter.

Evaluation

A large proportion of the limitations in this project come from the robot, 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 more controllable testing environment could allow for results that are more thorough.

The ideal design for the class using probabilistic filtering is to gather clusters of the areas with the highest probability (initially 0.5, as unknown areas) and approach 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 lack 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]
Available at: http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2D/Direction_to_Angle.php
[Accessed 18 March 2014].

MSDN (2014) *Thread.Abort Method.* [Online]
Available at: 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 Center (2007) *Determine if the point is in the polygon, C#.* [Online]
Available at: 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].