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 was heavily considered in the planning and implementation.

## Program structure

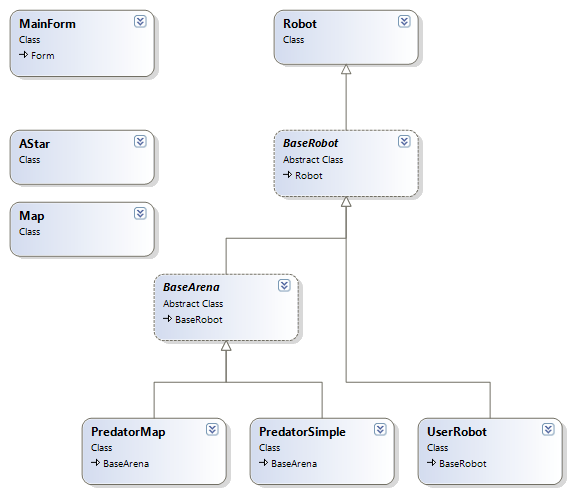
Due to the decision to create two separate implementations, there was careful deliberation regarding the good practice of reusing code. The program falls into three layers.

**Abstract ‘robot’ class**   
The topmost class. This holds all basic Rovio commands that is both found in the web API and useful in this project (such as receiving a camera image and moving). 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 be derived 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 is expected to be derived from the arena class, and can thusly use all data that is collected regarding developments in the environment.

The application would have been equally functional in this case 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.

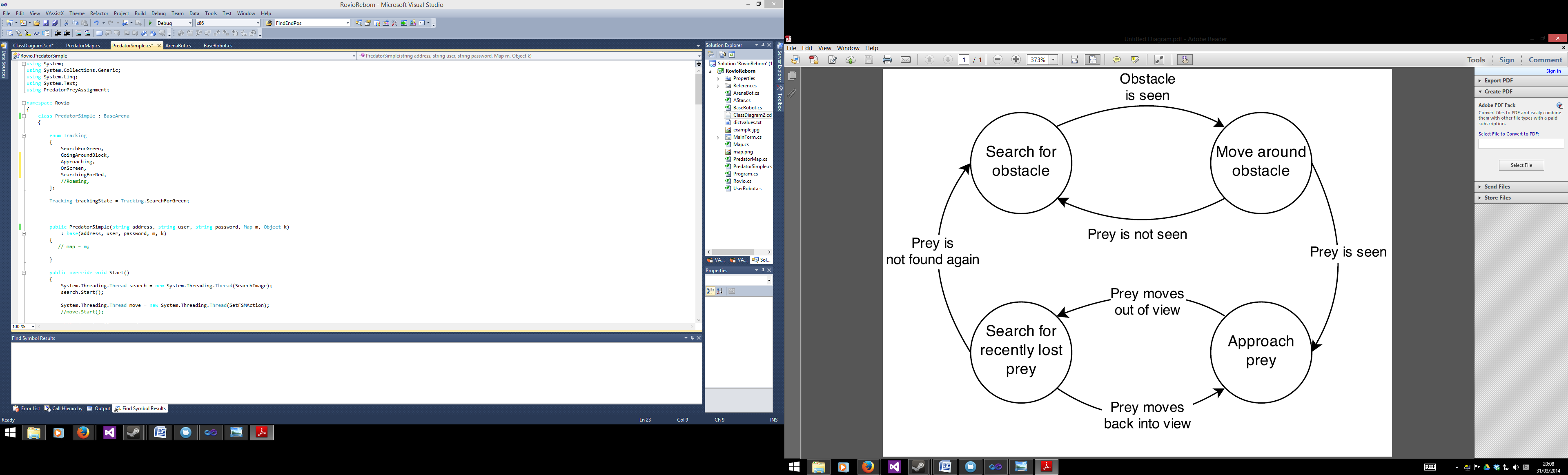
****

Figure 2 – 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.

# 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 introduces the (very likely) possibility that multiple commands will be simultaneously sent to the Rovio. 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.

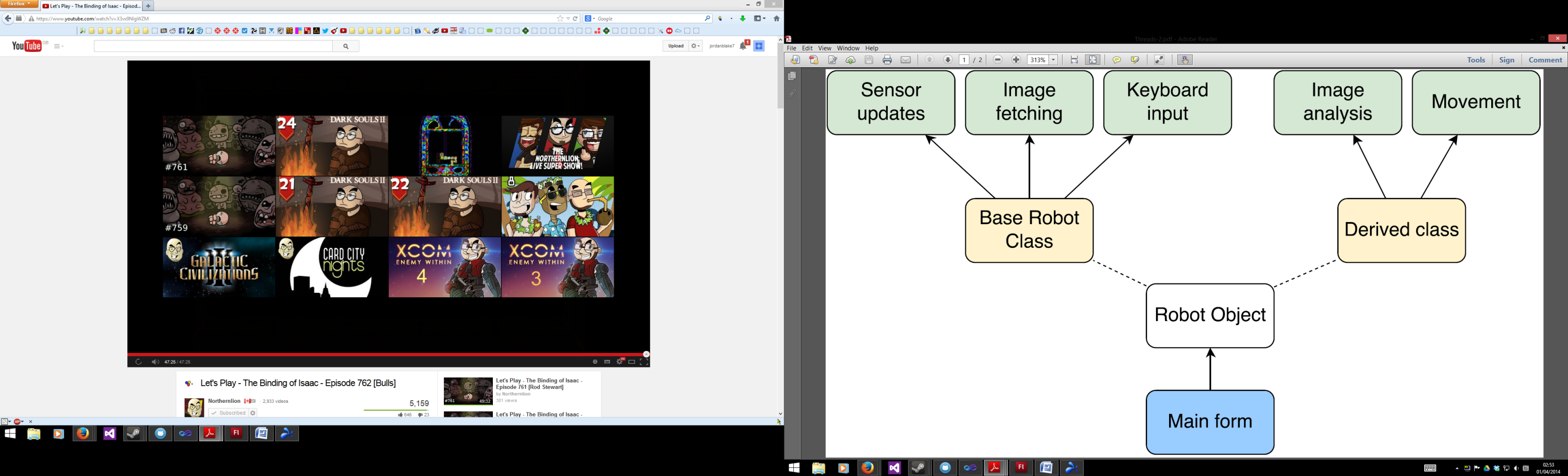
The main form creates a thread, which runs the base robot. From the base robot class, two threads run: one for receiving the camera image, and one for accepting keyboard input from the main form. The derived classes execute their own threads for searching through the camera image and for their own movement 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 3 - 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 the way that 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 reinitialisation 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 ‘Wait’ 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.

## Finite State Machine

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’.

**Initial approach** – move towards the obstacle until it is within a short distance.  
**Strafe around** – continue strafing until the block is out of sight (i.e. there is a clear path ahead).  
**Move forward** – Go forward by one metre (slightly more than the length of the obstacle).  
**Rotate** – Perform a 90° rotation around the obstacle.

If the prey enters view during any phase of the ‘move around obstacle’ state, 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.

# 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.