Robotics Practical Assignment – Predator & Prey

**Technical Design**

Structurally, the solution was implemented using a predator python class to keep the code modular, readable, and reusable for further improvements. This utilised class variables for the ‘ROS Topic’ publisher and subscribers used in its methods, as these are uniform across all predator instances. Also used were primitive instance variables, set and applied to control logic at key points throughout the implementation. This enabled interoperability between the topics and the data they sent and received during the predator’s runtime. The callback methods for these subscriber topics are the entry point for application code execution, as they call the methods to perform the location, detection, computer vision analysis, movement, and avoidance actions.

From an architectural standpoint the instance variables could act as ‘model features’, akin to those used during the ‘Plan’ robotics primitive. In this case, the predator would be considered as having a deliberative control architecture. However, as all environment information is obtained locally through sensors, is frequently overwritten, and is not used to construct a global model – meaning it isn’t used to plan necessary directives to achieve its goal – it ultimately uses a reactive approach. As such, different topic callbacks run concurrently (some using threading) and can directly affect one another as per the architecture’s “vertical task decomposition” (Hanheide, 2015).

**Player Behaviour – Predator**

This mode implements behaviour co-ordination using a finite state machine approach. When activated, it applies computer vision techniques using data received from the Turtlebot’s passive exteroceptive camera sensor and, based on this, utilises its mobile effectors and actuators to search for its prey. If the prey isn’t found in the same general vicinity the predator will repeatedly change its location and search again until it is. Upon detection, it will begin pursuit until it ‘tags’ it – using its tactile, proprioceptive bumper sensor to confirm the capture – and then notifies the user of its victory with a distinctive action. If it loses sight of the prey it will attempt to reacquire it, by searching in the prey’s last relative location on its camera sensor (left or right), increasing efficiency. If found it will resume pursuit, otherwise it will revert back to its search state. Should obstacles be encountered or introduced, the bot will circumvent them before resuming its current action, using active exteroceptive laser sensor data to determine the best avoidance action to take. This was added toward the end of development, as shown in the following figures.

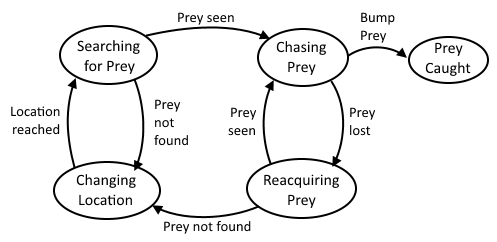
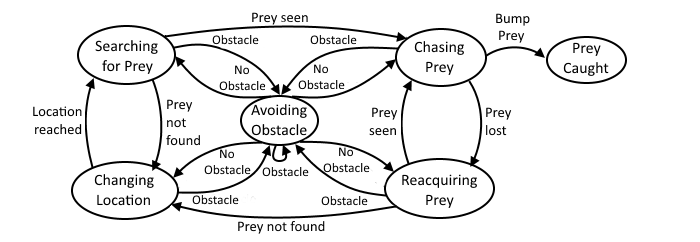
****

Figure 1 – Finite State Model (simple)

Figure 2 – Finite State Model (with Obstacle Avoidance)

****

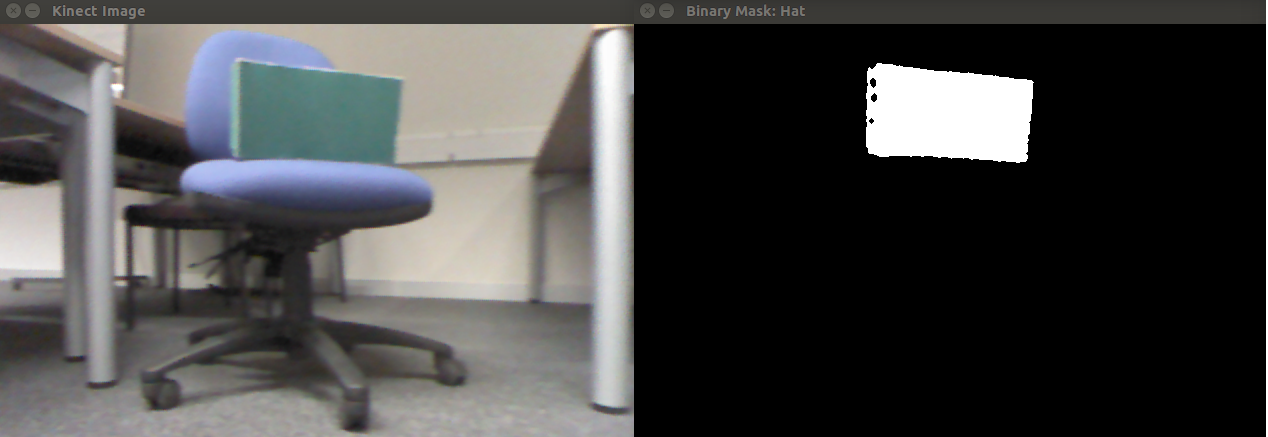
**Opponent Detection**

**Identification**

In accordance with the computer vision pipeline the camera image was acquired from the Kinect by subscribing to the ‘camera’ topic. It is then pre-processed by applying a 5\*5 Gaussian Blur to remove some noise, followed by format conversion and colour slicing. Due to its inherent performance benefits and its closer representation of how humans see colour (François & Medioni, 1999), the HSV colour space was used to define these ranges. Each channel had hard-coded upper and lower values taken from unaltered screenshots of the Turtlebot camera’s view of their green hat, used with openCV’s ‘inRange’ function to produce a binary mask.

Pre-processing then overlapped with segmentation to perform some common morphological operations; a double iteration ‘erosion’ followed by an ‘open’, both of which using a 3\*3 elliptical kernel. This image could now be used for viewing as in Figure 3 below, demonstrating its efficiency.

Figure 3 – Kinect Image vs. Binary Mask



A simple ‘BinaryImage’ class was added to hold the final binary mask once it was normalised, so it could be declared an instance variable and used throughout the code. Feature extraction then iterated through the binary mask to find (and save to ‘hat\_size’) the largest single object using openCV’s ‘findContours’, which if larger than a pre-defined value would be considered the prey. The ‘x’ value of the hat’s centroid was then found using openCV’s ‘moments’ function (K, 2012). This was used to determine the ‘last\_relative\_hat\_location’ relative to the centroid of the binary mask. It then decides if ‘is\_prey\_escaping\_view’ by comparing it to the left & right boundaries, as marked by red dotted lines on Figure 3. Now the predator’s behaviours included a ‘goal-oriented appetitive’ behaviour.

**Location**

A ‘search’ function was needed to locate the prey if it didn’t start in view of the predator’s camera. This used two main navigation strategies. Firstly, a local “Search” strategy so it could “move and recognise its goal” (Franz & Mallot, 2000). Secondly, a primitive global “Recognition-triggered response” strategy using the previously mentioned valriable: ‘last\_relative\_hat\_location’.

If the mask’s largest singular object – the green hat – contained less than a certain number of pixels, the ‘search\_for\_prey’ function started on a separate thread. It then constructed and published a ‘Twist’ object to the predator’s ‘cmd\_vel’ topic, setting it a relatively slow turning speed. This rotated it once fully on the spot, but still slow enough to retain high image quality and allow the computer vision to determine if the prey is found during. For the ‘recognition-triggered response’ navigation, if the prey was last seen on the right of the camera it would rotate right instead of its default left.

In doing the above, the predator was able to see all unobstructed areas in its general vicinity and – assuming it detected it – could proceed in pursuing its prey. Should the basic default search motion have failed in locating the prey, it triggers a Boolean flag that switches its motion from rotation to advancing forward for a set time. Once it’s reached its new location the flag triggers back and it reverts to rotation, now potentially able to see more of its environment. This concluded the ‘exploration/directional’ behaviour implementation.

**Pursuit**

Once located, the prey will be in at least partial view of the bot’s camera. With this image data the predator is able to make decisions about what to do next within its closed-loop control, using a ‘proportional controller’ alongside a ‘Braitenberg Aggression’ vehicle implementation.

Firstly, a custom function uses array indexing to split the binary mask into two vertical halves. It then passes it to a ‘determine\_movement\_velocities’ function that gets the sum of all pixels in each half. It then calculates a ‘normalised\_mean’ by dividing the difference in the two pixel counts by the total pixel count. The proportional controller in this closed loop control will output a normalised angular velocity ‘z’ value that is directly proportional to the intensity of the input image. This takes inspiration from the ’excitatory’ connections in the Braitenberg vehicle 2b (1986).

Now the predator needs to decide if it will go straight toward the prey or if it needs to correct its course because the prey is outside of the previously defined boundaries of the image. This uses the instance variable: ‘is\_prey\_escaping\_view’. If it is in the middle, this will be false and the predator will advance, ignoring the angular velocity previously calculated. However, if true it will correct its course by passing both calculated linear and angular velocities to a ‘Twist’ object to be published.

**Capture**

Now that the Turtlebot has detected, located, and is in pursuit of its prey, it should be able to catch it. To do so it used the Kinect camera’s emulated Laserscan, an active exteroceptive sensor, the ROS topic for which it subscribed to upon creation. Once the prey is in close visual range, the capture conditions will check the predator instance’s most recently updated ‘hat\_size’ variable. If it falls within pre-set ranges found through testing, ‘is\_prey\_in\_catching\_range’ is set to true.

One final sensor is utilised to determine whether the predator has caught the prey; the Bumper (I Heart Robotics; Peter Tran, 2014). This passive sensor has very low CPU requirements and therefore proved very responsive in testing, so was an ideal fit for a final ‘is\_prey\_caught’ check. Once bumped it will back up slightly and, if ‘is\_prey\_in\_catching\_range’ is true, it calls the final method. This makes the predator rotate 360 degrees left and then 360 degrees right, to make it clear that it has caught the prey. In the event that it is bumped and there are an insufficient number of hat pixels visible, it will simply back up slightly and continue running.

The custom method set up to handle all movement (‘publish\_twist\_msg’) makes sure to set safe parameters for movement speeds. The max linear and angular speeds being 0.3 and pi, respectively. If the method parameters try to go over these, the method will reset them to the max speeds.

**Obstacle Avoidance**

**Identification**

The laser data was also used here. This was due to its lower CPU cost, suitability for the role, and comparable simplicity when compared with self-localisation, mapping or odometry-based solutions.

This is handled in the same callback used for Prey Catching in the previous section. It gets the array of ranges back, finds the ‘minimum\_distance’, and if it is less than 0.6 it will take action. The predator’s decision about which action to take is based on which side the ‘minimum\_distance’ lies, relative to the robot. The ranges array was split into two sub-arrays using python’s ‘len()’ function to count the number of elements, then indexing based on this amount. If the ‘minimum\_distance’ value was in the set of values pertaining to the left of the robot, ‘current\_obstacle\_location’ would be set to “LEFT” and vice versa.

**Action**

With sufficient data for its purpose, assuming the network does not cause significant lag, the bot will take immediate, overriding action to avoid any obstacle encountered within its ‘minimum\_distance’. When this occurs and the hat isn’t in catching range, it will set ‘is\_collision\_imminent’ to true (a flag used throughout to prevent over-publishing of messages), as well as disabling course correction (also preventing it from sending conflicting Twist messages). It then performs pre-set avoidance actions.

If it has not had to avoid any obstacles thus far, it will reverse away from it for 1 second, rotate for 3 seconds in the opposite direction to that which is stored in ‘current\_obstacle\_location’, then advance for 2 seconds. This also increments it’s ‘avoid\_count’ value and resets ‘is\_collision\_imminent’ and ‘is\_course\_correction\_disabled’ to ‘false’.

If it has had to avoid more than 5 obstacles rapidly (i.e. the counter hasn’t been reset due to inability to reach the search/pursue functions, since it’s been ‘stuck’ avoiding multiple obstacles) it will enter an escape function. This function temporarily reduces the ‘minimum\_distance’ threshold to give it a better chance of escaping from complex environments such as tight spaces or areas with chair legs.

**Testing & Evaluation**

**Functionality & Performance**

Testing the efficiency of the implementation was done by running the functional components together in different scenarios, repeating a uniform number of times (10), and observing the error frequency.

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Expected outcome** | **Error frequency** |
| Predator put in open space with prey | Prey found, chased, caught | 0% |
| Predator placed facing wall and activated | Avoids wall, begins searching for prey | 10% |
| Predator placed in tight corner with single navigable exit | Avoids obstacles, enters escape mode | 60% |
| Predator shown prey, prey moved backward around corner | Sees prey, searches in direction of corner, advances past corner, searches, catches prey | 20% |

While the results showed positive results for most tests, they highlighted a significant error margin for the obstacle avoidance functionality. It did enter escape mode as intended, however it bumped into obstacles and did not handle the event as intended. This is possibly due to the prescribed motions for avoiding, whereby the predator reverses, rotates, and advances for set times. It receiving no data about its tight surroundings during these closed-off movements would explain the bumping into the obstacles that occurred during 6 out of 10 of the tests in that scenario.

Accuracy of system components that had the potential of returning noisy, incomplete or inaccurate datasets was then tested. This had the added benefit of helping to ascertain the ideal level of error handling required to further improve the predator mode in the future.

**Laserscan distance averages (10 measurements)**

|  |  |  |
| --- | --- | --- |
| **Wall distance (actual)** | **Wall distance (average scan reading)** | **Standard deviation** |
| 0.25m | 0.131m | 0.136 |
| 0.50m | 0.489m | 0.009 |
| 0.75m | 0.852m | 0.097 |
| 1.00m | 1.301m | 0.218 |

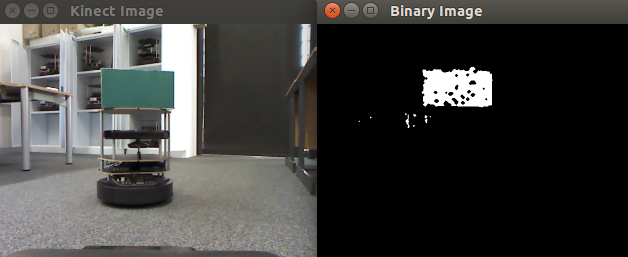
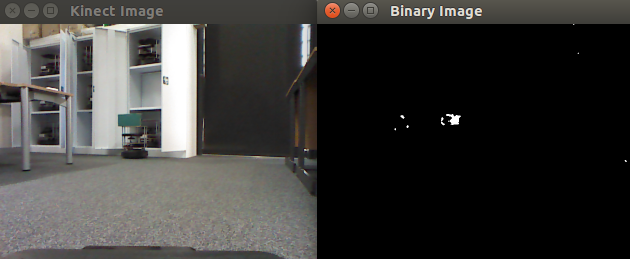
Since the laser is emulated using the Kinect camera it’s to be expected that the distance data would not be as accurate as a dedicated laser sensor. Interestingly, however, the closest reading was not the best. The most accurate was from 0.5m away suggesting that the ‘NaN’ readings in the range array (viewable upon printing to the console) are increasingly frequent the closer it gets to very close objects, affecting the readings significantly.

**Binary Image data - prey detection at different ranges (10 measurements)**

|  |  |  |
| --- | --- | --- |
| **Prey distance** | **Hat pixel difference average (Kinect vs Binary)** | **Standard deviation** |
| 1m | 158 | 56 |
| 2m | 297 | 72 |
| 3m | 690 | 201 |
| 4m | 1016 | 2130 |

Given the necessary programming time and processing involved in implementing dynamic colour learning, the performance of the HSV colour slicing used on this predator was considered acceptable. The results show an exponential increase in the difference in hat pixels across the raw image and binary mask when the prey distance increases. The standard deviation value for 4m came out surprisingly high as well, which may have been caused by an increase in noise at this range. As such it was treated as an anomaly. Figure 4 shows examples of the black box test data.

Figure 4 – Kinect/Binary pixel differences Near vs. Far



**System parameters, limitations & Improvements**

Network latency from too many requests on a shared network was a big issue during development. Multiple robots running at once severely affected the performance and efficiency of the system and while every effort was made to eliminate this variable in testing, it was out of developer control. Loading the code directly onto it may be an option in the future. CPU requirements of some functions, specifically the computer vision section, will have had an effect on the performance too to some extent. It’s entirely possible that the frequency of processor-intensive method calls could be reduced without affecting system performance, but this would need to be properly tested on a private network to ensure accurate data.

Making the code more efficient by further separating it out into segregated, modular components would go far in increasing reusability. From there more advanced threading could be implemented, for instance to more clearly define and separate the states defined in Figures 1 and 2.

Sensor inaccuracy is another issue that’s largely unavoidable. Odometry measures, for instance, can produce notoriously inaccurate data without proper localisation and mapping implemented alongside it. The addition of odometry to Laserscan or vision algorithms could also reduce margin for error in that case, so it would be something to consider to further the work on this predator bot. A similar combination could be used if a longer-range distance estimation was required, to identify the prey’s exact location or direction. Finally, as robust as the colour slicing used to process the images may be, support for different lighting, longer distances, and more effective noise reduction have all been tested and identified as having room for improvement.

References

Braitenberg, V., 1986. *Vehicles: Experiments in synthetic psychology.* 2nd ed. s.l.:MIT press.

François, A. R. & Medioni, G. G., 1999. Adaptive color background modeling for real-time segmentation of video streams. *Proceedings of the International Conference on Imaging Science, Systems, and Technology,* 1(121), pp. 227-232.

Franz, M. O. & Mallot, H. A., 2000. Biomimetic robot navigation. *Robotics and autonomous Systems,* 30(1), pp. 133-153.

Hanheide, M., 2015. *Robotics Control Architectures.* [Online]   
Available at: https://blackboard.lincoln.ac.uk/bbcswebdav/pid-923414-dt-content-rid-1873725\_2/courses/CMP3641M-9037M-1415/CMP3641M%20Robotics%20lecture%206%20-%20Control%20Architectures.pdf  
[Accessed 26 February 2015].

I Heart Robotics; Peter Tran, 2014. *Cornell College - Move.py.* [Online]   
Available at: http://people.cornellcollege.edu/smikell15/MAX/code/move.py.html  
[Accessed 20 March 2015].

K, A., 2012. *Official Python Tutorial Codes - Moments.* [Online]   
Available at: https://github.com/abidrahmank/OpenCV2-Python/blob/master/Official\_Tutorial\_Python\_Codes/3\_imgproc/moments.py  
[Accessed 1 March 2015].