Braitenberg Vehicle Demonstrating a 'Fear' of Spherical Objects

Jamie A. Lord[1]

[1]*Lincoln School of Computer Science, University of Lincoln, UK*

jlord@lincoln.ac.uk

Abstract— **This paper presents a Braitenberg vehicle that demonstrates a fear-like behaviour towards spherical objects detected using a visible light camera. The solution utilises the Microsoft Kinect's ability to measure distances using infra-red markers to calculate the 'level' of fear to demonstrate and the speed of which to retreat from the identified sphere(s). Additionally, a basic collision avoidance system was implemented to enable the platform to organically find targets in it's surroundings by ensuring that it does not become trapped in one location and remains hunting for spheres to avoid. Performance of the system in the ROS (Robot Operating System) simulator was promising but performance of the sphere-detection subsystem was less accurate when deployed to real hardware. The final system did however perform well considering a limited amount of testing using real-world visual input data. In conclusion, the final system was moderately capable of determining spherical objects on real-world data and the capability of the system to avoid objects was highly-successful and could easily be enhanced to create a robust solution to ensure continued operation of a robotic platform in a previously unknown environment.**

Keywords— **braitenberg vehicle, robot operating system, fear, spherical object**

I. Introduction

Implementing a fear-like behaviour is a common Braitenberg experiment; the original experiment [1] utilised light as a simple input that could easily be controlled and sensed using the technology available at the time. Now that more advanced sensors are available it is possible to detect complex structures using image capturing systems.

The ability to detect spherical objects within a field of view and then react based on the orientation and distance has many real-world applications; an example that would require this behaviour is a robotic system that plays dodge ball. Such a robot would need to identify and respond to visual input within a small timeframe if it were to be capable of playing the game. More serious applications could include automatically tracking spherical features on a production line, for example ensuring the location on bottle caps on a conveyor belt – the mathematical representation of the location of the sphere could be used to effect any form of behaviour.

The hardware platform used for this experiment was the TurtleBot 2 [2], this small platform includes a Microsoft Kinect sensor and a low-energy personal computer that provides enough processing power for real-time image processing tasks whilst also running on battery power [3]. The robot uses the Robot Operating System [4] to provide a flexible framework, built on top of a typical Linux operating system, that enables rapid development of software using standardised libraries and feature-sets. ROS's publisher-subscriber model was used in this implementation to make the sphere-finding subsystem output available to other software components.

The system was developed primarily in the simulator and was ultimately tested on real-world hardware. Due to resource limitations, testing on physical hardware could not be conducted until a late stage of development. This is likely to have been a primary factor in the lack of performance of the sphere-detection system in the real-world. The imperfections in a live video stream from a real camera are impossible to fully simulate; as such, the system's ability to detect spheres was typically too sensitive when deployed to the robot. This could however be corrected with more development using the physical robot or by incorporating a more realistic environment in the simulator, this is further discussed in the following sections.

II. Related Work

The paper 'Emotional Generation Model for Autonomous Mobile Robot' [5] outlines a method for computing emotions and storing them in an emotion vector. Maeda focuses on sadness, joy and anger; the vector defined stores a combination of these values in a three-dimensional space, enabling complex emotions to be defined.

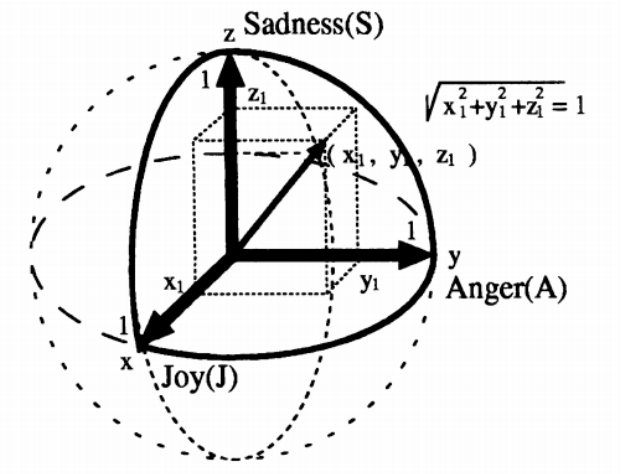

Fig 1. Basic emotion vector

Basic robots are used that include a light sensor that provides a normalised input to the emotion generation system. Using only a 3D vector, other emotions were determinable including disgust, fear and love. The final results of the study show that a variety of emotions are easily implementable, even on a robot with very simple sensors and actuators.

Subsequently, the notion of emotional flexibility could be applied to the solution developed here. The robot could demonstrate a number of emotions toward spherical objects rather than just fear. This idea could be explored further by also incorporating other values such as the distance to the sphere(s) as well as their number and size – this complex behaviour could have a wide range of real-world applications.

Detecting circles in an image is a core component of this research, enhancing the algorithms used could lead to superior performance or more accurate results. The paper 'Fast Circle Detection using Gradient Pair Vectors' [6] highlights both the issues with classical implementations of Hough transformations as well as how they can be rectified. The proposed detection method uses a pair of gradient vectors to determine the edge of the circle(s) compared to the background portion of the image.

The Hough detection method is typically storage and processing intensive this can have major disadvantages for real-time applications such as this one. Their solution is noted as having eighty times the performance of the CHT algorithm. The magnitude of this performance improvement when deployed to limited or embedded hardware would be of particular interest considering the platform used for this research.

A core function of this system utilises the depth data taken using a Microsoft Kinect sensor array; improving the characteristic of this data could vastly improve the overall performance of the complete system. Peasley and Birchfield [7] propose an algorithm to detect and avoid obstacles for mobile robot platforms. The implementation demonstrates the versatility of the Kinect sensor as well as some of its limitations – "depth readings for specular surfaces and for objects that are too close to the sensor" [7] proved difficult to accurately measure.

The largest improvement the paper demonstrates over ROS's handling of the Kinect depth data is the ability to provide readings beyond a single horizontal plane. This makes determining the height of specific objects possible, something can cannot be done with the current implementation.

III. Methodology

The system was designed to leverage ROS's inherent publisher-subscriber model to handle movement and visual data-flows. The entire system has been implemented in a single class due to the limited feature-set of the complete system. A vision and depth callback method was used to handle the input streams from the ROS subscribers.

Fig 1. outlines the basic structure of the class and includes arrows denoting the directional flow of data passed between methods.

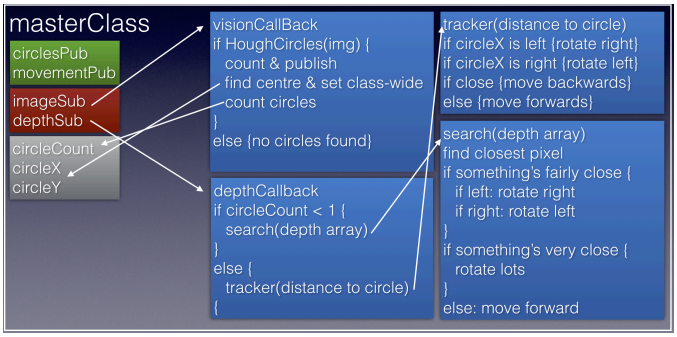

Fig 1. Complete system method overview

Of note is the relationship between the *depthCallback* method and the *search* and *tracker* methods. The depth data is used to determine if an object is close enough to the robot that avoiding action should be instigated. Some class-wide variables are also included in the diagram to illustrate how they are used across multiple methods to share common data.

The *search* method is passed a 1D depth array by the appropriate callback method and and proceeds to determine where in front of the robot the closest object(s) are. The following pseudo code outlines the algorithm used.

```
def search(self, depthArray):
    closestDistance = 1000
    index = 0
    for i in range(0, 640): #check for closes object
        if closestDistance > depthArray[300, i]:
            closestDistance = depthArray[300, i]
            index = i
    #move avoiding object if close enough
    twister = Twist()
     #Turn away depending on which side of the image its in
    if closestDistance < 1:
        if index < 320:
            twister.angular.z = -1
        if index > 319:
            twister.angular.z = 1
    #move back if too close to something else
    #keep moving in a straight line
    if closestDistance < 0.6:
        twister.angular.z = -5
    else:
        twister.linear.x = 0.2
    self.movementPub.publish(twister)
```

Fig 2. Search method

The method analyses the entire array of depth information to determine the index of the closest pixel. A *Twist* object is then initialised to send the appropriate movement actions to. It is then determined whether the pixel that is closest to the robot is on the right or left, the robot is then turned in the opposite direction to avoid collision. Although crude, this algorithm has proved acceptable when running on a real TurtleBot and only struggles when the robot gets into a tight corner with an object on both the right and left. In this situation, the algorithm could be enhanced by turning the robot without moving forward until there is nothing directly in front of it.

Determining the location of a circle in the input image is implemented using OpenCV's *HoughCircles* method [8]. This feature accepts an image – in this case a frame from the robot's camera when a new one becomes available – and

returns an array containing the central coordinates and radii of any circles detected depending on the specified parameters. The parameters used in this solution include the minimum and maximum radius of the returned circles – these were arbitrarily set; a future enhancement of the system could be to enable these settings to be set dynamically depending on the characteristics of the image being processed.

If any number of circles are found, then the *tracker* method is called; this uses the coordinates of the centre of the identified circle to firstly rotate the robot so that the circle is in the centre of the camera field of view and then moves the robot backwards.

IV. Evaluation

The solution was initially developed in the ROS simulator to the point that a working solution was ready for deployment to a physical TurtleBot. Moving the solution to a real robot with imperfect inputs and motor movements proved to demonstrate the difference between a simulated world and the real thing. The camera inputs were comparatively noisy and produced a larger number of false-positives than the simulator; see Fig. 3. Real-world testing was conducted using a green, circular object that was distinct in colour to the background.

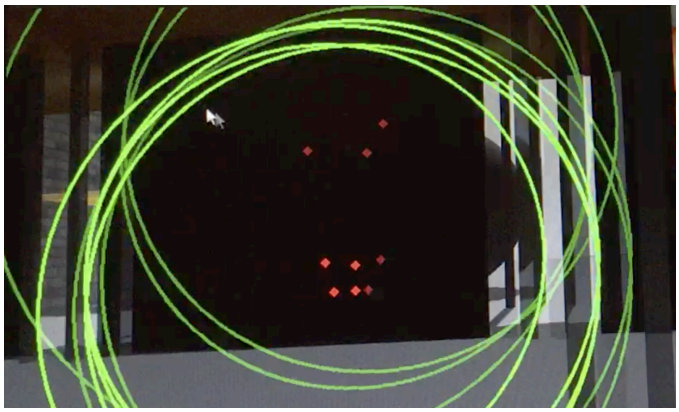

Fig 3. Example of miss-classification of camera input

When testing the system, it became apparent that the number of circles being detected from the feed was too high; the number of false positives made movement of the robot impossible. This highlights the potential issues when developing a robotic system in a simulator without using real-world data. However, it should be entirely possible to remedy this issue by measuring the features of the circles across multiple frames to stop circles being detected in a single frame. This would however have the effect of slowing down the response time of the robot, but this may be required to get the system working with real-world data.

The robot's ability to navigate around complex objects however was successful – the TurtleBot was capable of avoiding table legs easily.

There are numerous system parameters that could be modified to enhance the overall system. For example, changing the parameters used with the *HoughCircles* function call could improve the performance of the circle detection without implementing the proposed multi-frame analysis function.

In the real world, the solution is prone to over-sensitivity and thus fails to accurately identify the features of concern. The object avoidance aspect of the system is however very successful and with further enhancements could yield very strong results.

The overall performance was strong – the TurtleBot includes a powerful x86 computer that is easily capable of processing the camera's output in real-time without dropping frames. The camera worked well in typical 'office' lighting but may struggle in lower light situations. A potential improvement to the system could leverage the Kinect's ability to generate an image using infrared light outputted by the device itself. This would mean the robot could operate in complete darkness, although it is still possible that objects with certain light-absorbing properties could disrupt the quality of the vision input.

V. Conclusion & Future Work

The final solution utilised the TurtleBot hardware well although the performance of the sphere sensing subsystem was less than desired in the real-world. However, the collision avoidance system was very successful considering the simple nature of the algorithm used and delivered results that where unexpectedly strong. With only minor modifications, this system could be made to be very robust for environments where the lighting is strong and the surface is flat.

The circle detection function used could be enhanced to increase accuracy on the robot; this could be achieved by analysing and storing the circles from previous frames to minimise errant identifications. If a circle wasn't identified within a sequence of frames, then it would be discarded; a window of movement could also be set to further reduce false identifications. Implementing this feature would simply require storing a pre-determined set of circle data for each frame in a buffer that is constantly overwritten when new camera images become available from the ROS subscriber.

Enhancing the collision avoidance system could also improve the overall performance of the robot. The robot occasionally got trapped when entering a corner; identifying this situation algorithmically would be simple and lead to a rotation of the robot such that it is no longer facing into the corner.

In conclusion, this implementation of a Braitenberg vehicle using ROS on a TurtleBot has been successful in the simulator with some components working well in the real-world. With minor enhancements and more time developing for the real robot, the solution could perform very well. Fixes to the discovered issues were identified that could make a future implementation robust and perform well using real-world data.

REFERENCES

[1] V. Braitenberg. Vehicles: Experiments in Synthetic Psychology 1986.

[2] W. Garage. Turtlebot. Website: Http://turtlebot.com/last Visited pp. 11-25. 2011.

[3] B. Gerkey and K. Conley. Robot developer kits [ros topics]. Robotics & Automation Magazine, IEEE 18(3), pp. 16-16. 2011.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng. ROS: An open-source robot operating system. Presented at ICRA Workshop on Open Source Software. 2009, .

[5] Y. MAEDA. Emotional generation model for autonomous mobile robot. KANSEI Engineering International 1(1), pp. 59-66. 1999.

[6] A. A. Rad, K. Faez and N. Qaragozlou. Fast circle detection using gradient pair vectors. Presented at Dicta. 2003, .

[7] B. Peasley and S. Birchfield. Real-time obstacle detection and avoidance in the presence of specular surfaces using an active 3D sensor. Presented at Robot Vision (WORV), 2013 IEEE Workshop on. 2013.

[8] Anonymous (). Hough Circle Transform — OpenCV 3.0.0-dev documentation 2015.