CAMSHIFT Tracking

David Wood, Vale of Leven Academy
Mark Jenkins & Gordon Morison, School of Engineering and Built Environment GCU

Abstract

This report presents an implementation of the CAMSHIFT algorithm for fast, efficient and computationally inexpensive colour-based object tracking. Originally tracking a single colour, now tracks multiple objects of varying colours in real time on desktop. Multiple selection methods are implemented to make selecting objects to track easy and quick. Paths of objects can also be visualized and drawn in the colour of the object being tracked. The whole system was implemented for use on embedded devices such as the Beagleboard xM and performance was improved using NEON Instruction Sets for these ARM-based devices.

Introduction

Image Processing is a vast field encompassing many different disciplines, the main focus of this report is the applications of Tracking with Image Processing and the algorithms that make this possible. Our system makes use of an algorithm known as CAMSHIFT, from the Computer Vision Face Tracking For Use in a Perceptual User Interface by Bradski [1] which uses the most prominent areas of a particular colour to track an object - our aim in brief was to use the algorithm to track multiple objects of different colours in as close to real time as possible.

Many tracking algorithms exist, but often elaborate methods use various hypotheses or feature detectors which are far too computationally expensive. Therefore this focuses on colour-based tracking algorithms, such as CAMSHIFT, which can track a given colour in the presence of noise, other colours and movement while still remaining fast and efficient. However, even many colour based tracking algorithms are far too computationally expensive (and therefore slower at any given CPU speed), especially for use on embedded hardware, such as Beagleboards.

Furthermore, the system was designed for ease of use, where the tracking of multiple objects of varying colours would be easy to set up, without needing to recompile. The system also makes use of multiple different methods of selection, each suited to different shapes of object, it is also made easy to toggle between these selection methods in a way that makes sense and is intuitive. The system is also not limited to having only one selection method at any given time, at runtime each object can be selected using whichever method suits that object best.

Aims

The system aims to employ the CAMSHIFT algorithm to track colours by hue, it was then adapted to have the capability to track multiple objects, of varying colours, at the same time with an intuitive and user-friendly system of interaction. We achieve this by assigning each object being tracked an index, which is mapped to the number row on the keyboard, this provides a familiar way to toggle the tracking of multiple objects without the difficult memorisation of shortcut keys.

The system aims to be be able to follow the object as it moves around in the frame and support full 360° rotation, the system theoretically can also support an object leaving and reentering the frame, so long as the colour doesn't change.

Originally the system used hard coded hue values, but was expanded to encompass the ability to select the hue dynamically from the video feed at any time, at first this was done via selection, where the user would draw a box over the object and using a histogram, the most prominent colour would be determined. However, as the scope of the system expanded, 2 more selection methods were added, referred to as Histogram and Point. Histogram selection creates a histogram from the whole frame, and then lets the user select the colour from that histogram. Point selection is the simplest of the three and allows the user to double click to select a radius around the pointer for selection.

The selection system aims to allow the user of the system to easily select the object for tracking at runtime with whatever method makes sense for the current object. Where the object is not rectangularly shaped, methods such as Histogram or Point selection make more sense. Therefore, selection methods have to be flexible and easy to use, so it's simple to switch method at any time, for any object.

Set-up

Development on the system was performed on an Ubuntu 12.04 Virtual Machine, and the code was written completely in C++, it makes heavy use of the Open Computer Vision (OpenCV) [2] library, and through use of the OpenCV library, certain common operations were abstracted away such as getting capture frames. OpenCV also provided many classes, such as *Mat*, that were used extensively when handling image data as these classes contained many functions that sped up development by not having to write our own code that's been written many times before.

During development, the Linux GNU Compiler Collection (Linux GCC) was used to compile the system for testing, however, for deployment on the beagleboard [5], the Cross GNU Compiler Collection (Cross GCC) was used with a specific version of the OpenCV library built for the ARM architecture of the beagleboard. We then made use of OpenSSH to transfer cross-compiled binaries to the embedded board for testing.

While CAMSHIFT aims to use as few system resources as possible and be computationally efficient, the performance of the code on the embedded board was unacceptable, and so the NEON Instruction Set for ARM [4] processors was employed, these replaced the standard C++ math functions with assembly optimised NEON functions resulting in a 100% increase in frame rate.

Tracking

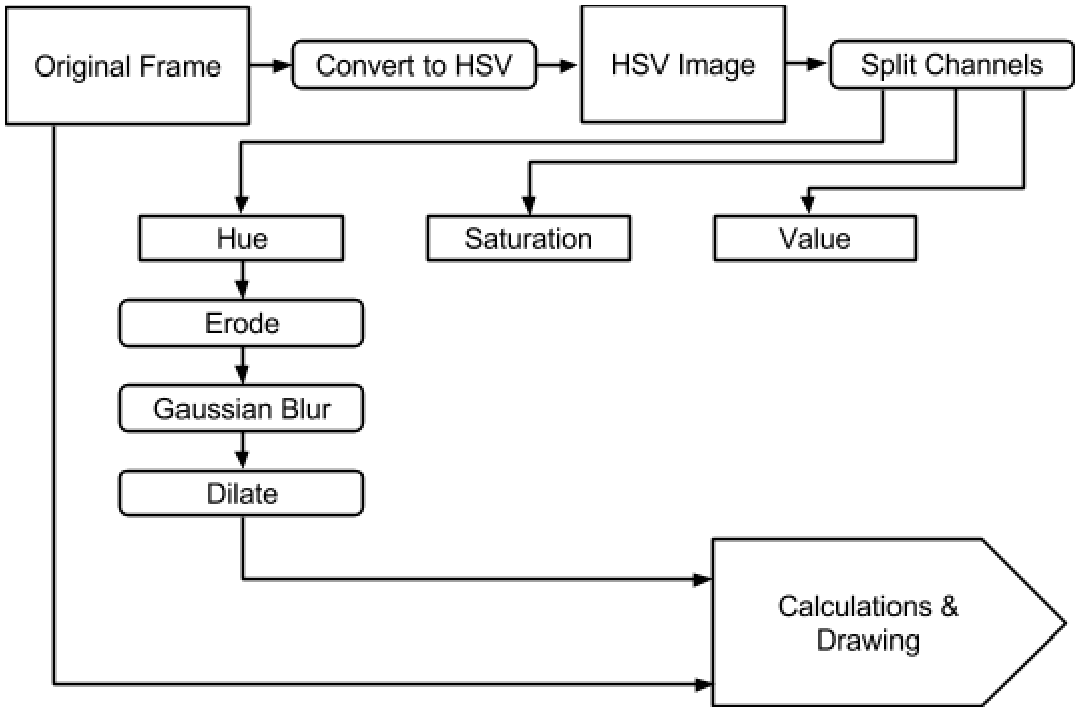


Figure 1: Diagram of stages involved.

In order for use, CAMSHIFT begins by converting the current frame or image from RGB (Red Green Blue) colour to HSV (Hue Saturation Value) colour. HSV differs from RGB, where in HSV, the range of colours is in the Hue value, which is a spectrum of available colour, as shown below.

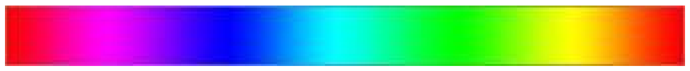


Figure 2: Colour Spectrum.

The Saturation value is the tint of the colour, where a minimum value of 0 would turn the whole spectrum white, and a max value of 255 would retain the full colour spectrum shown above. The Value value is the shade of the colour, where a minimum value of 0 would turn the whole spectrum black, and a max value of 255 would retain the full colour spectrum shown above.

Note: OpenCV only uses a range of 0 to 180 for the hue value, as opposed to the more standard 0 to 255.

Note: OpenCV interprets the hue value as BGR instead of RGB, where a value of 0 would typically be red, in OpenCV, a hue value of 0 is blue.

CAMSHIFT uses only the hue channel of an image, as the hue value contains the actual colour itself, and when tracking colour, any additional computation on other channels would be a waste.

Therefore, the next step is to split the HSV frame into it's channels: Hue, Saturation and Value. Using the hue value, the image is thresholded using our minimum and maximum HSV values, this removes everything that does not fall within the range, leaving a black and white image with two values, 0 for the black, and 1 for the white. The white areas represent where parts of the image are between the hue range supplied.

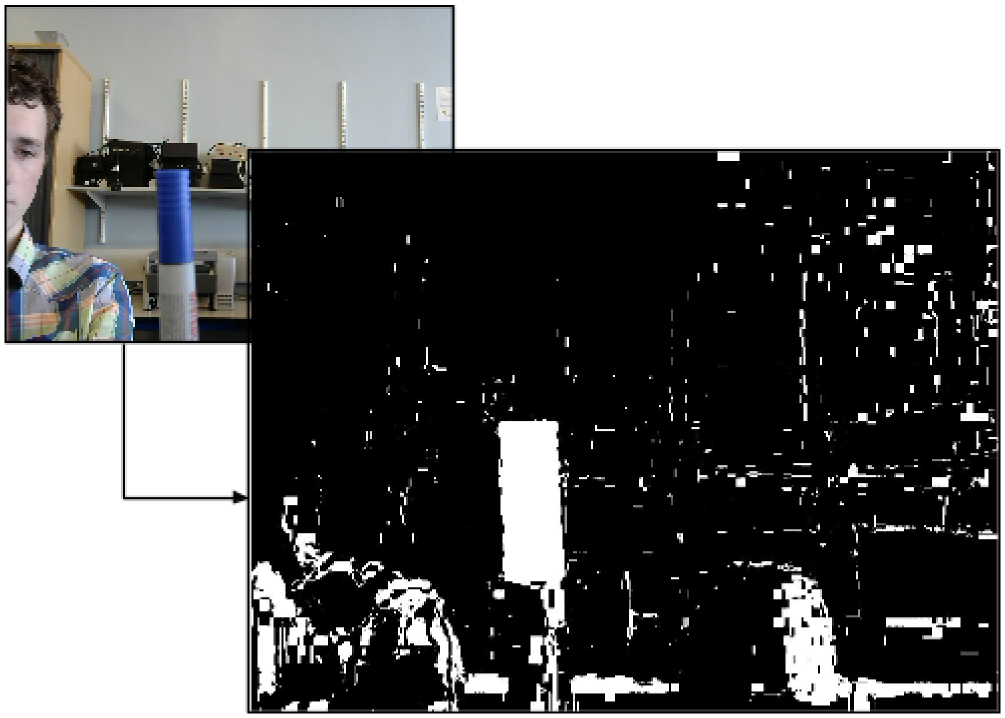


Figure 3: This shows the difference between the original image and a thresholded image, notice the artifacts.

This operation often leaves artifacts on the thresholded image, or very small dots of white around the object from things like reflections or smaller background objects that may have been within the hue range, this would cause issues for further calculations in the CAMSHIFT algorithm, therefore a technique known as erosion, which removes any small artifacts and some small parts of the edges of any objects.

Since further calculations will use the thresholded area to calculate lengths and widths, it is important that the thresholded object is at roughly the same size as the object in the original frame, due to the erosion however, this is not the case, and rectifying the problem is a two step process. The first of which being applying a gaussian blur, this removes any rough edges created by the erosion. The second of which is Dilation, where the thresholded range is expanded slightly, this returns the thresholded area to around the size of the object in the original frame but without the artifacts.

Then, it is necessary to find the largest contour, as this is the object that will be tracked. In this implementation, finding the largest contour was done using the *findContours* and *contourArea* functions of OpenCV.

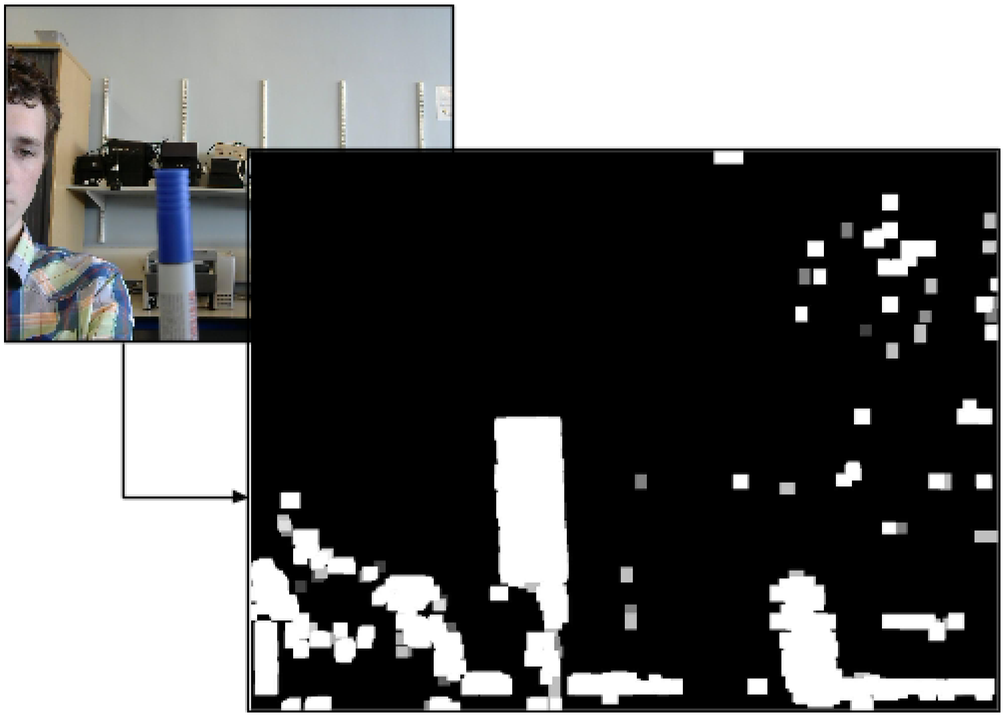


Figure 4: This image shows the final state of the thresholded image before finding moments.

Calculations

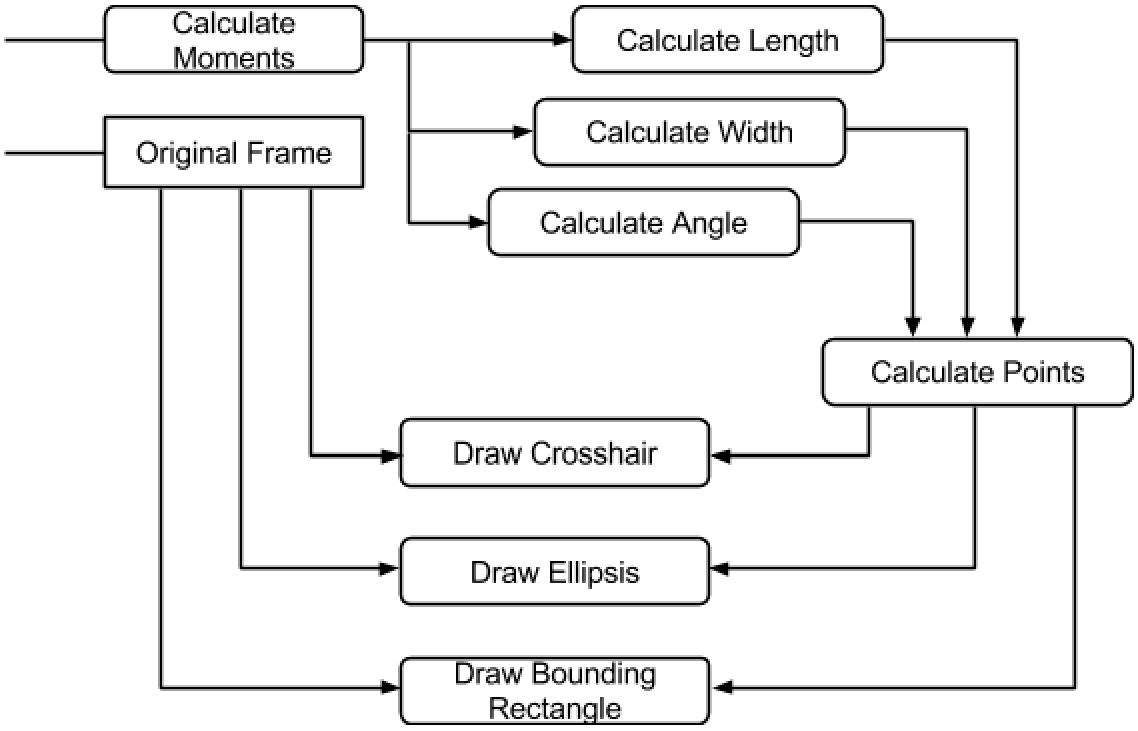


Figure 5: Diagram of stages involved.

The next stage is finding the Moments of the largest contour. Moments are calculated and used to determine the width, length and angle. The first moment calculated is m_{00} , this is done by adding the value of all the pixels together, therefore m_{00} is the sum of all pixels. The next moments calculated are m_{10} and m_{01} for the x and y respectively. m_{10} is calculated by summing the product of each pixel and the x coordinate value for that pixel. Likewise, m_{01} is calculated by summing the product of each pixel and the y coordinate for that pixel. For example, in an object that is very wide (lots of x values) compared to it's height, therefore m_{10} will be larger than m_{01} . These three points can then be used to calculate the centre point of the object.

Note: In this implementation, all of the math relating to moments are abstracted away into the *moments* function of OpenCV.

$$m_{00} = \sum_x \sum_y I(x, y)$$

Find the zeroth moment.

$$m_{10} = \sum_x \sum_y xI(x, y) \qquad m_{01} = \sum_x \sum_y yI(x, y)$$

Find the first moment for x and y.

$$x_c = \frac{m_{10}}{m_{00}} \qquad y_c = \frac{m_{01}}{m_{00}}$$

Calculate the mean search window location (the centre).

Second-order moments are used when calculating the width, length and angle of an object. The second order m_{11} moment can be calculated as shown:

$$m_{11} = \sum_x \sum_y xyI(x, y)$$

The second order m_{20} and m_{02} can then be calculated as follows:

$$m_{20} = \sum_x \sum_y x^2 I(x, y) \qquad m_{02} = \sum_x \sum_y y^2 I(x, y)$$

Using the moments, the m_{00} moment being the centre point, the next step is to calculate the angle, width and length of the object being tracked, this information can be used to display the overlay on the object.

$$a = \frac{m_{20}}{m_{00}} - x_c^2 \qquad b = 2\left(\frac{m_{11}}{m_{00}} - x_c y_c\right) \qquad c = \frac{m_{02}}{m_{00}} - y_c^2$$

$$\theta = \frac{arctan(\frac{b}{a-c})}{2}$$

Calculate the angle of rotation from the vertical in radians.

$$l = \sqrt{\frac{(a+c)+\sqrt{b^2+(a-c)^2}}{2}}$$

Calculate the length of the object.

$$w = \sqrt{\frac{(a+c)-\sqrt{b^2+(a-c)^2}}{2}}$$

Calculate the width of the object.

Note: In C++, there are two variations of the arctan function, $atan(\frac{b}{a-c})$ and $atan2(b,\ a-c)$, the former function returns an angle that varies from 0° to 45°, and then from -45° to 0° in 8 quadrants, the latter returns an angle that varies from 0° to 90°, and then from -90° to 0°, in practice the latter function was significantly easier to work with.

Note: The lengths and widths returned by the formulas above are from the centre point to the edge, therefore they should be doubled to represent the whole length or whole width.

$$\theta_d = \frac{\theta}{\pi}\times 180$$

Calculate angle in degrees.

In Face Tracking, the above equations result in the head roll, length and width. CAMSHIFT therefore gives an efficient, simple to implement algorithm that tracks four degrees of freedom.

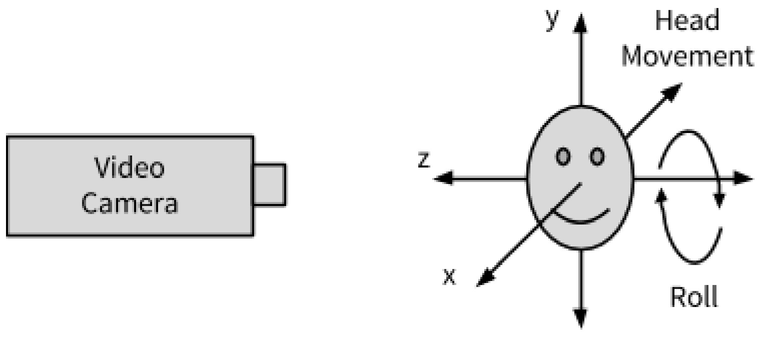


Figure 6: Diagram of four degrees of freedom.

Using width, length and centre point, the topmost (v_1), bottommost (v_2), rightmost (h_1) and leftmost (h_2) points can be determined. Using Trigonometry and Pythagoras, the horizontal and vertical steps can be determined, then when added/subtracted from the centre point result in the top, bottom, left and right points.

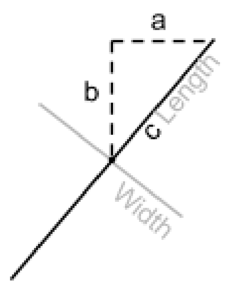


Figure 7: How steps are calculated.

When calculating the points, the longest side, c, is always either the length or the width, a triangle is then created from the longest side to determine the step on the x and y axis.

$$\theta_i = (0.5 \times \pi) - |\theta|$$

Calculate the inner angle.

Calculating Vertical Points

$$v_c = x_c$$

$$v_a = v_c \times sin(\theta_i)$$

$$v_b = \sqrt{v_c^2 - v_a^2}$$

Calculating Horizontal Points

$$h_c = y_c$$

$$h_a = h_c \times sin(\theta_i)$$

$$h_b = \sqrt{h_c^2 - h_a^2}$$

Note: When using OpenCV drawing functions, the $(0, 0)$ coordinate often refers to the top-left point (as with most computer graphics applications), so when calculating the top, it is subtracting the step, not adding.

θ **< 0:**

$$v_1 = (x_c - v_b, y_c - v_a)$$

$$v_2 = (x_c + v_b, y_c + v_a)$$

$$h_1 = (x_c + v_b, y_c - v_a)$$

$$h_2 = (x_c - v_b, y_c + v_a)$$

θ **> 0:**

$$v_1 = (x_c - v_b, y_c + v_a)$$

$$v_2 = (x_c + v_b, y_c - v_a)$$

$$h_1 = (x_c + v_b, y_c + v_a)$$

$$h_2 = (x_c - v_b, y_c - v_a)$$

Using these points, it is trivial to draw crosshairs on the object (for example, using OpenCV's line function), or draw an ellipsis around the object. It may be preferable to increase the h_a, h_b, v_a and v_b points slightly so as to have some padding around the object.

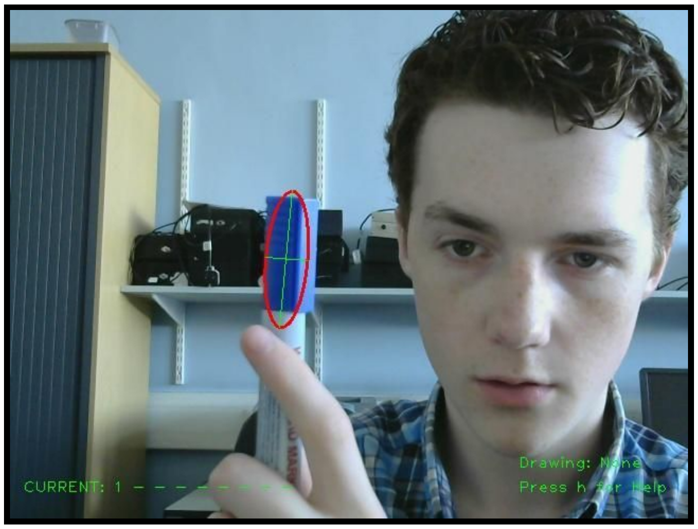


Figure 8: The image above shows the ellipsis and crosshair drawn using the points above.

However, when drawing a bounding box around the object, the top-left (c_1), top-right (c_2), bottom-left (c_3) and bottom-right (c_4) points are needed, which require more calculation.

$$a = |h_2.x - x_c| \qquad b = |h_2.y - x_y|$$

Let a equal the distance between the centre and the left side on the x-axis.
Let b equal the distance between the centre and the left side on the y-axis.

θ < 0:

$$c_1 = (v_1.x - a,\ v_1.y + b)$$

$$c_2 = (v_1.x + a,\ v_1.y - b)$$

$$c_3 = (v_2.x - a,\ v_2.y + b)$$

$$c_4 = (v_2.x + a,\ v_2.y - b)$$

θ > 0:

$$c_1 = (v_1.x - a,\ v_1.y - b)$$

$$c_2 = (v_1.x + a,\ v_1.y + b)$$

$$c_3 = (v_2.x - a,\ v_2.y - b)$$

$$c_4 = (v_2.x + a,\ v_2.y + b)$$

Note: OpenCV doesn’t have a rotated rectangle function, so an alternative method is to connect each point with the line function.

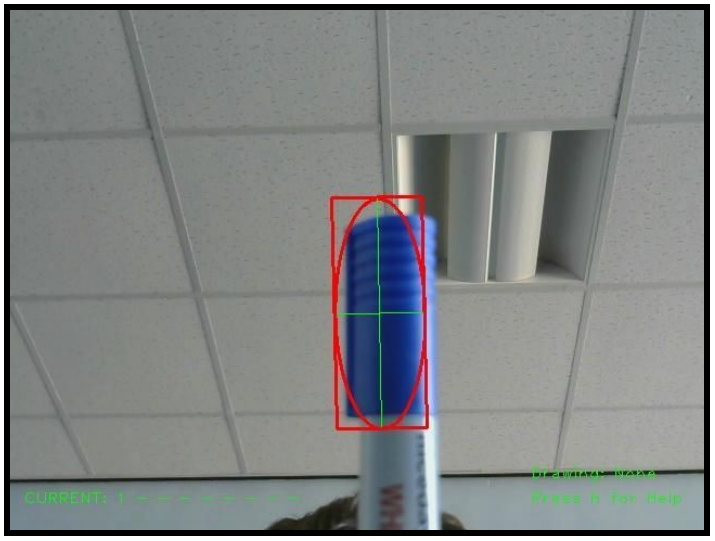


Figure 9: This image shows the bounding box being drawn using the 4 points calculated above.

Selection Methods

As previously mentioned, there are 3 main selection methods implemented in the system: Point, Selection and Histogram. All three methods use a Histogram to find the most prominent colour from the selection area.

Also for each method, a single value is returned from each method, be it Histogram, Point or Selection, this value is then padded with a configurable value to create a hue range. If the base hue value from the histogram was $baseHue$ and the padding was $hueOffset$, then to calculate the minimum and maximum hue:

$$\begin{aligned} minHue &= baseHue - hueOffset \\ maxHue &= baseHue + hueOffset \end{aligned}$$

Histogram-based selection takes the whole frame and generates a Histogram of hue values from which the user can then select via double clicking the colour to track.

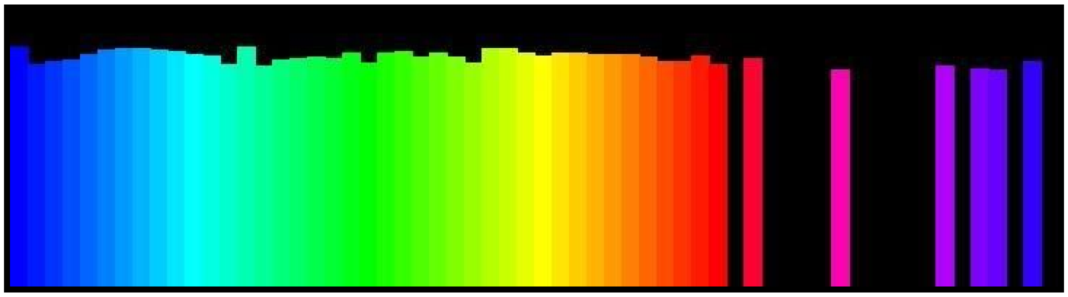


Figure 10: The above image is an example of one of the output Histograms from the *Histogram* selection mode.

As with the tracking implementation, only the hue value is used when generating the Histogram, in this implementation, the Histogram is calculated using OpenCV’s *calcHist* function when specified the channels, range for each channel, and number of bins.

Since only the hue channel is used, the function is supplied with the channel index, 0, and the range of the hue channel, 0 to 180. Currently there are 60 bins, resulting in a 3 values per bin.

Note: When drawing the Histogram to the screen, some scaling is applied so that it is a reasonable size, for example, instead of each bin being 1px wide, they are scaled to 10px wide.

When drawing the histogram, the rectangles are drawn using OpenCV’s functions in HSV colour, the whole histogram is then converted to RGB colour when drawn.

In *Histogram* selection, a window is shown with a Histogram like that shown above, the user double clicks to select a colour. Using the x-coordinate of the mouse click, division is used to remove the scaling applied on the shown Histogram, and then multiply up by $\frac{180}{no.\ of\ bins}$ to get the actual hue value.

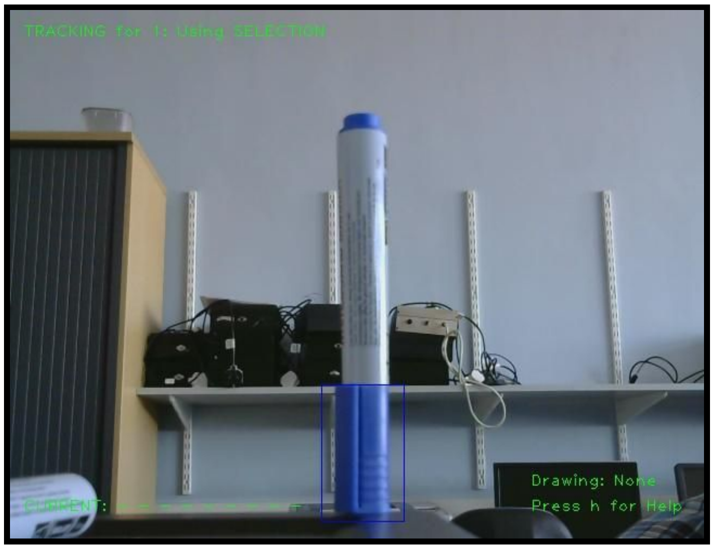


Figure 11: The above image shows the bounding box being drawn when in Selection mode.

Selection mode was originally the only method of selection, and was designed to be simple and intuitive to use. This mode works by allowing the user to draw a box over the object on the screen (using OpenCV’s mouse events), the area is then isolated and a histogram is generated. Whichever colour is most prominent in the histogram is then used as the base value for the hue range.

Point selection is arguably the easiest and most commonly used selection method, it works by using coordinates of a double click (from OpenCV’s mouse events) to generate a small area around the click which is then used in the histogram to generate the base value for the hue range.

Multiple Objects

During the development of the system, it was noted that a few common elements were required to track any given object. These elements were:

- Minimum Hue
- Maximum Hue
- Centre Point
- Length
- Width
- Rotation Angle
- Top/Bottom/Right/Left points

The system was then adapted to support the tracking of multiple objects, this involved creating a common structure that could be used in one of C++'s container classes to be iterated over when drawing and running calculations.

Objects are added and removed easily by using any of the number keys from 1-9, each of which refer to an index of a collection (typically the index will deincrement by 1 so that it starts at 0, like the collections themselves), the system could support more, obviously at the expense of frame rate as the number of objects being tracked increases therefore so does the number of computations, but is currently only limited by the space on the keyboard.

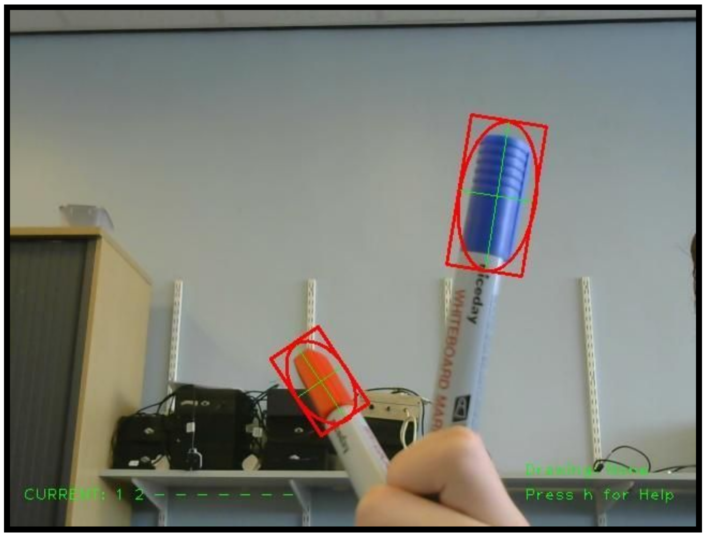


Figure 12: The above image shows the system tracking two objects of different colours simultaneously.

Originally a *vector* collection was used, as unlike a *linked list* or *doubly linked list*, a *vector* supports fast random access. A *vector* was also chosen over the other Sequential Containers, such as *deque* or *array* because, unlike *deque*, the container had to support fast insertion in the throughout the container, not just the end and, unlike *array*, had to be flexibly sized. However, in the end, a *vector* was not chosen to contain the tracking objects, as due to the nature of interaction, where objects could be inserted or removed at any index, an Associative container was a better choice. In this case, a *map* where the key was an integer was used, as it supports fast random access, but also fast insertion and deletion at any given index. A *map* was chosen over other Associative containers for various reasons: *map* was chosen over a *set* because in a *set*, the value is the key, which wouldn't be applicable here; *map* was also chosen over *multimap* and *multiset* because unique keys were necessary; *map* was also selected over the various unordered associative collections such as *unordered_map*, *unordered_set*, *unordered_multimap* and *unordered_multiset* because the benefits these bring would not be of use in this application and would only add overhead from the use of hash functions.

```
for (map<int, TrackingObject>::iterator item = objects.begin(), end = objects.end();
item != end; ++item) {
	updateObject(item->second, frame);

	if ((item->second).moments.size() > 0) {
		runCalculations(item->second);

		drawOverlay((item->second), frame);
		(item->second).lastTop = (item->second).topPoint;
	}
	if (item->first == debugIndex) {
		drawDebugInformation(frame, item->first, item->second);
	}
	if (item->first == drawingId) {
		if (startDrawing) {
			item->second.centres.push_back(item->second.centre);
		}
	}
}
```

Figure 13: The above code example shows how the loop functions in the current system.

Before the addition of tracking multiple objects, there were several variables passed throughout the system that held the required information from one single object, with the addition of the single structure to hold the information, parameters for many functions were cut down significantly, and had to be changed to access properties of the structure. However, the only major changes were in the main loop of the algorithm where it was necessary to iterate over the *map* of tracking objects.

Drawing Object Trail

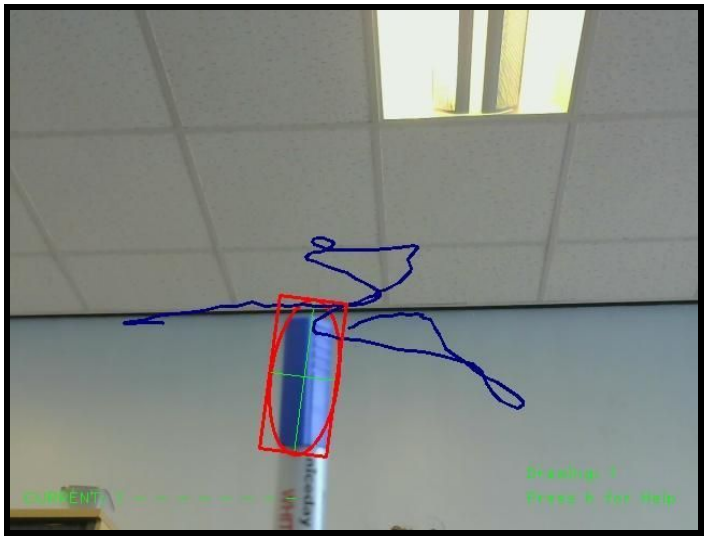


Figure 14: The above image shows an example of the path of the object's centre being drawn.

The system was also adapted to include the ability to draw the path of the object as it moves, in the colour of the object. This was achieved by adding a *vector* of former centre points to the structure of tracking objects. This *vector* could now be iterated over in the drawing functions to connect each point with a line.

While a *vector* was used, another sequential container type such as a *linked list* could have been used and would have been more effective, when logging former centre points, they are only added to the end of the container, and when being drawn, they are only iterated over sequentially, not randomly. Therefore the *vector* could be replaced with a *linked list*, and would likely improve performance slightly.

The system was designed to draw the path in the colour of the object that is being tracked, this is done by using the minimum and maximum hue value contained in the object.

$$avgHue = (minHue + maxHue) \div 2$$

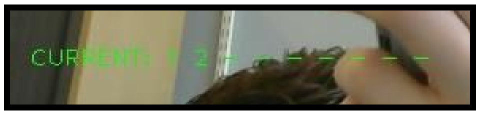
We calculate the average hue as this will be the the value being drawn. However to be drawn, unlike the histogram calculation, the whole frame cannot be converted to RGB after the fact, so it is necessary to convert the hue value into an RGB colour for being drawn.

Note: In this system, the hue value was converted to a RGB colour by filling a 1x1 OpenCV *Mat* with the HSV colour, then converting that whole *Mat* to RGB with OpenCV's *cvtColor* function. The value was then taken from the converted *Mat* to be drawn.

Only one object's path is ever drawn live at any given time, however, until cleared, paths from other objects are persistent.

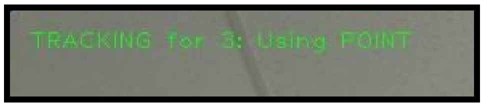
User Interface

Much of this section has been touched on briefly elsewhere, but usability is a major aim of the system. For that reason, the User Interface has been designed to be intuitive and easy-to-use.



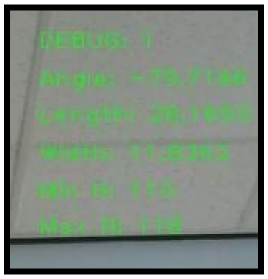
```
CURRENT: 1 2 - - - - - - - -
```

The UI for selection of multiple objects was designed to be easy to use, it also aimed to quickly display how many objects are being tracked.



```
TRACKING for 3: Using POINT
```

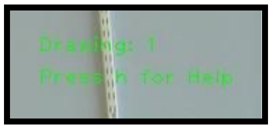
Selection UI had to clearly show which index was being assigned to and which



```
DEBUG: 1
Angle: -79.7146
Length: 28.1693
Width: 11.6363
Min H: 110
Max H: 118
```

Debug UI was used exhaustively in development, this view shows clearly all the relevant information for any given object.

The Debug view can be toggled using a specific key, it iterates over the *map* of objects being tracked plus one, this way the UI can be hidden when appropriate.



```
Drawing: 1
Press h for Help
```

When drawing the path of an object, only one given object can be drawn at a time, this is then persistent till cleared. This UI was designed to easily show the current object being drawn.

The system also has the ability to easily take screenshots quickly, designed primarily to be used for this report, this system is simply a keybinding, and doesn't have an interface component. Shown below is the final UI element, a help screen, toggleable with *h*, this shows all of the current features and the relevant keybinding.

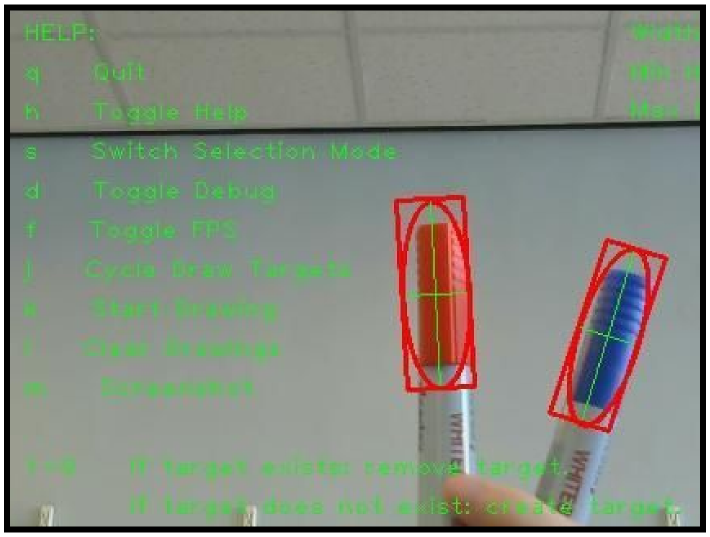


Figure 15: Example of UI.

Embedded Implementation

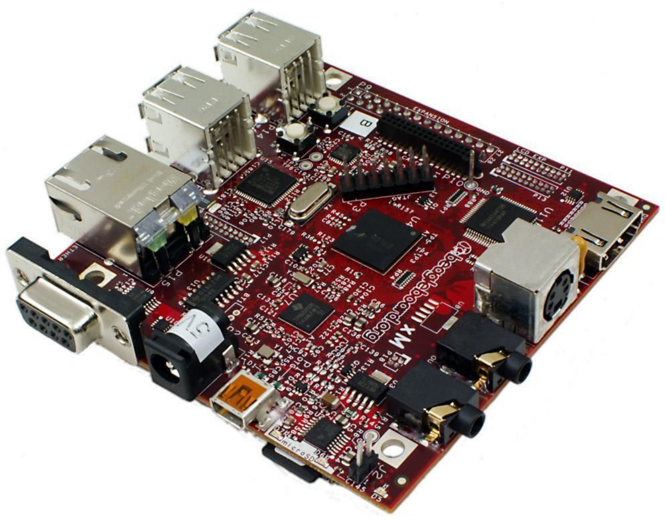


Figure 16: Beagleboard xM

During the implementation of the system, it was cross-compiled to run on embedded systems such as Beagleboards [5], as pictured above, as these systems are typically less powerful than traditional desktop computers, and so performance of the system on embedded boards is often worse than on traditional desktop computers, as would be expected. To run the system on the board, cross compiling was required using specific versions of the OpenCV library compiled for ARM processors.

The beagleboard used for this implementation was running Angstrom [3], a linux distribution designed for embedded devices. The board had a 1Ghz processor and was connected via HDMI to the monitor for output.

To improve performance of the system, the NEON Instruction Set [4] was used to replace C++'s standard math functions, these functions were assembly optimised specifically for the ARM architecture and resulted in a 100% increase in frame rate.

Conclusion

This implementation uses CAMSHIFT, a simple computationally efficient color-based tracking algorithm, designed to work as well as other tethered trackers or more expensive vision systems, even in the presence of noise. Our implementation succeeds in tracking objects with full 360° rotation, no matter how far from the capture source, it also works when the object has left and reentered the frame.

The system also succeeds in providing an easy-to-use system for tracking multiple objects of multiple colours at any given time. It provides multiple easy-to-use selection methods for selecting objects to track, each suited to different shapes of objects.

Currently the system struggles on embedded implementation due to the computational nature of the algorithm, while it performs better than other algorithms of it's type, this performance will be seen to improve as embedded boards become more powerful due to new innovations in hardware. There are also potential improvements to be made with regards to tracking multiple of the same colour, a limitation of the current system which consistently selects the largest object that fits the hue range, therefore never being able to track the same colour at the same time, the system could be modified to sort objects in the hue range by size, and then tracking the 2nd or 3rd object if hue ranges overlap significantly.

Reference

[1] Bradski, Gary R. "Computer vision face tracking for use in a perceptual user interface." (1998).
[2] OpenCV "http://opencv.org/"
[3] Angstrom "http://www.angstrom-distribution.org/"
[4] NEON "http://www.arm.com/products/processors/technologies/neon.php"
[5] Beagleboard "http://beagleboard.org/"