ROCO318 – MACHINE VISION

Elliott White | 10467243 | MEng Robotics | 22nd December 2017

CONTOUR-BASED OPENCV APPLICATION

Contents

The Task:

1. Develop a Contour-Based or Hough-Based OpenCV application that can detect rectangles in a live webcam stream.
2. Use the colour of the rectangle to reject all matches except the test target. This can be done by training the software to recognise a particular colour or by programming in a selected colour upper and lower ranges.
3. Add range data derived from a calibration process, then estimate the distance of the live target.
4. Improve the robustness of the algorithm or allow it to operate over a wide distance range.

My Solution – an Overview:

To be able to detect rectangles in a live webcam feed I will be using a contour-based algorithm. I have chosen this, so the algorithm can detect any shape and is not restricted to shapes with straight edges. This could also be disadvantage as the algorithm may detect many contours and therefore must do more image processing than a Hough-based algorithm. Before any processing is done, any images or webcam feeds need to be converted from RGB to HSV. This is because we don't just want to look at the colours, but also the saturation and luma. This allows the algorithm to be much more robust with changing lighting conditions and shadows.

The first part of the calibration of the algorithm is determining the focal length of the camera, and using this to determine the distance of the given target. The focal length can be determined from knowing the actual width of the target, the perceived width in pixels, and a known distance from the camera.

$$F = (P \times D) / W$$

If the focal length is known, we can now calculate the distance of the target in a live feed by taking measurements of the perceived width of the target. This approach is based on triangular similarity.

$$D = (W \times F) / P$$

To do this calibration, I will give the algorithm a still image stored on the hard drive. This means the calibration will always be the same every time the algorithm is run and means I don't have to hold up the target a set distance away from the camera every time the algorithm is run. The calibration process consists of:

- Loading the saved image
- Applying a HSV mask using predetermined values based on what is best for the lighting conditions in the image
- Detecting contours
- Drawing a box around the largest contour (which should always be the target)
- Apply the focal length calculation give the actual width and perceived with of the target

Once this has been done, the algorithm will be able to calculate the distance the target is from the camera in a live feed.

To train the algorithm to learn the desired colour of the rectangle, we need to give it upper and lower limits on the HSV values. This tells the algorithm to only look at pixels that are within these ranges and will mask out all other pixels. To determine the ranges that are required, a window of the webcam feed will show along with six sliders. These sliders can be moved and will set either the upper or lower bounds of each of the HSV parameters. The effect of these bounds will be displayed on the live webcam feed, so the user can see how the ranges affect what the algorithm will detect.

Once the user is happy, the ranges of the HSV parameters will be passed back to the main algorithm where the algorithm enters a continuous loop. This loop consists of:

- Reading in a frame from the camera
- Convert the image into HSV colour space
- Apply a mask from the previously determined HSV ranges
- Find contours
- Approximate the number of edges for all contours
- Add text to the window to label the shapes
- Find the biggest contour and approximate the number of edges
- If it has 4 edges, then draw a box around it
- If 5 frames have passed, then update the HSV values
- Repeat

To make the design more robust, I have decided I will dynamically update the HSV parameter ranges every 5 frames. This means that if the lighting conditions change or a shadow is cast over the target, the algorithm should be able to still detect the target by changing the HSV parameters.

Updating the HSV parameters consists of:

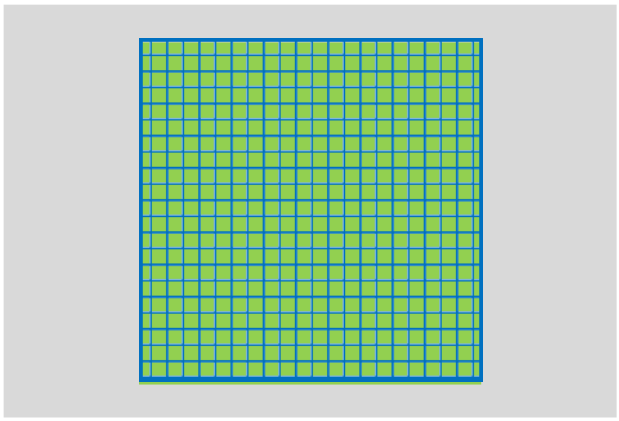
- Initialising variables
- Adding Gaussian Blur to the HSV image/frame passed from the main loop
- Creating an empty image
- Drawing the box contour of the detected target onto the empty image
- Filling in the contour with blue
- Finding the (x,y) positions of all pixels that are blue
- Creating a list of HSV intensities of all the pixels in the positions of the detected blue pixels from the HSV image/frame
- Calculating the average HSV values and the difference between the minimum and maximum of all values
- Creating HSV parameter boundaries based on the difference between the minimum and maximum HSV values
- Returning the average HSV values and the new boundaries back to the main loop

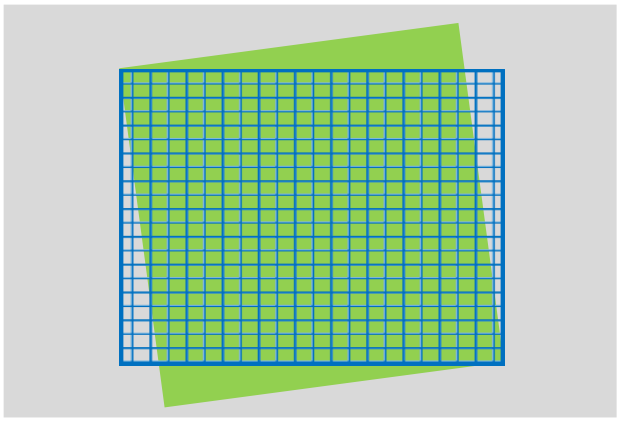
The main algorithm can then use these new HSV values and boundaries to adjust the masking and better detect the desired target. This method of updating the HSV values was not my first idea. I had originally tried to use the box contour coordinates as a boundary, and look at HSV intensities of the pixels within that box by looping over all the rows and columns. This initially worked, but I found that any rotation of the target caused pixels that weren't the target, to be read. This was due to me using

the top left and bottom right points of the box to define the boundaries of which pixels I would be looking at.

Target:

Pixels to be looked at:



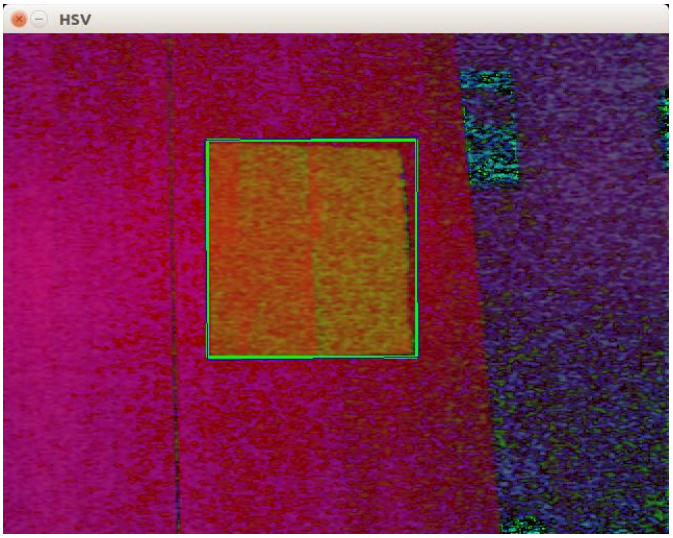


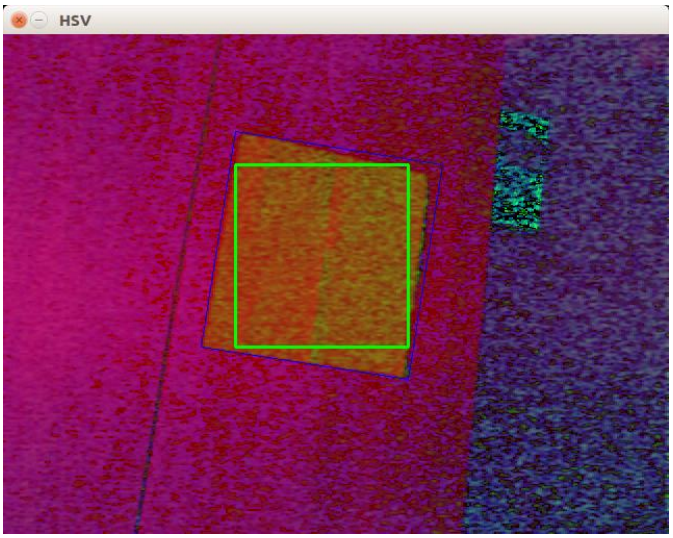
As we can see, any rotation of the target would change the boundaries and the pixels that weren't the target would be read. This caused the average HSV values to massively skew and caused the algorithm to completely lose the target. To rectify this, I decided to try and scale the box depending on the orientation of the target. This meant checking the corners of box contour and comparing them with each other to see if the values were bigger or small than each other. From this I was able to create a new box that would always be within the target, and then I could look at pixels within this box to get average HSV values. I found this to work better, but the more the target was rotated, the smaller the box got. This meant I was only looking at a fraction of the target and the calculated HSV values would not reflect the HSV values of the whole target.

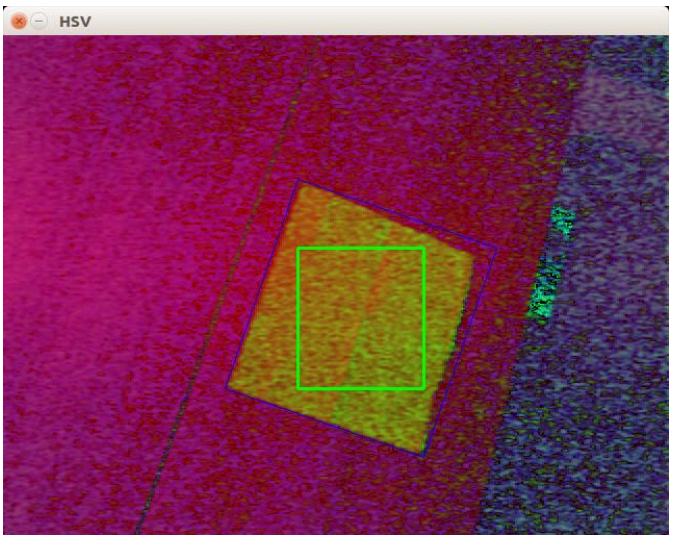
I will embed the source code on the following page.

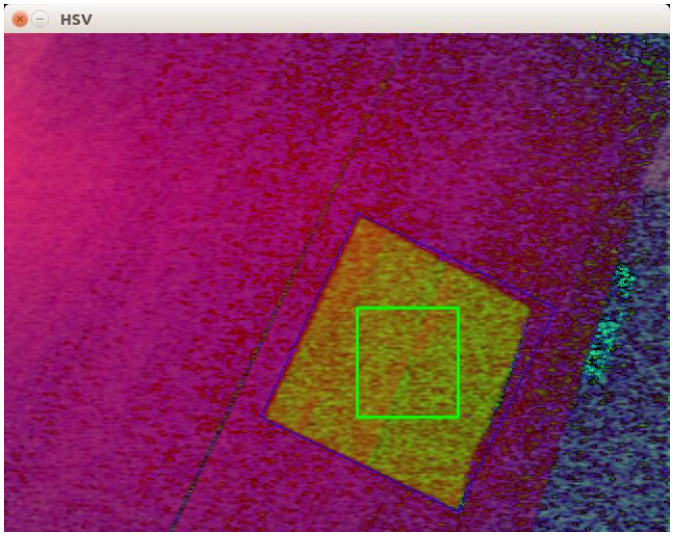
```
1.  def runScript(cap, hsvImage, box):
2.
3.      pixelCounter = 0
4.
5.      totalHue = 0
6.      totalSat = 0
7.      totalVal = 0
8.
9.      averageHue = 0
10.     averageSat = 0
11.     averageVal = 0
12.
13.     topLeft = box[2]
14.     topRight = box[3]
15.     bottomLeft = box[1]
16.     bottomRight = box[0]
17.
18.     blurHSV = cv2.GaussianBlur(hsvImage, (5,5), 0);
19.
20.     if topLeft[0] > bottomLeft[0]:
21.         newLeft = topLeft[0]
22.     else:
23.         newLeft = bottomLeft[0]
24.
25.     if topRight[0] > bottomRight[0]:
26.         newRight = bottomRight[0]
27.     else:
28.         newRight = topRight[0]
29.
30.     if topLeft[1] > topRight[1]:
31.         newTopRow = topLeft[1]
32.     else:
33.         newTopRow = topRight[1]
34.
35.     if bottomLeft[1] > bottomRight[1]:
36.         newBottomRow = bottomRight[1]
37.     else:
38.         newBottomRow = bottomLeft[1]
39.
40.
41.     cv2.line(hsvImage, (newLeft, newBottomRow), (newLeft, newTopRow), (0,255,0),thickness=2, lineType=8, sh
    ift=0)
42.     cv2.line(hsvImage, (newLeft, newBottomRow), (newRight, newBottomRow), (0,255,0),thickness=2, lineType=8
    , shift=0)
43.     cv2.line(hsvImage, (newRight, newBottomRow), (newRight, newTopRow), (0,255,0),thickness=2, lineType=8,
    shift=0)
44.     cv2.line(hsvImage, (newRight, newTopRow), (newLeft, newTopRow), (0,255,0),thickness=2, lineType=8, shif
    t=0)
45.     cv2.drawContours(hsvImage, [box], -1, (255, 0, 0), 1)
46.
47.
48.     #for 1st row to max row of box
49.     for row in range(newTopRow+15, newBottomRow-15): # Bin the first and last 15 rows and columns
50.         #for 1st column to max column of box
51.         for column in range(newLeft+15, newRight-15):
52.             pixelCounter = pixelCounter + 1
53.             rowSelect = row
54.             if(row > 479):
55.                 tmp = row - 479
56.                 rowSelect = row - tmp
57.             columnSelect = column
58.
59.             s = blurHSV[rowSelect, columnSelect]
60.
61.             totalHue = totalHue + s[0]
62.             totalSat = totalSat + s[1]
63.             totalVal = totalVal + s[2]
64.
65.
66.     if pixelCounter != 0:
67.         averageHue = totalHue / pixelCounter
68.         averageSat = totalSat / pixelCounter
69.         averageVal = totalVal / pixelCounter
70.
71.
72.     print('Total Hue = ', totalHue, 'Pixel Counter = ', pixelCounter)
73.
74.     print('H = ', averageHue, ' S = ', averageSat, ' V = ', averageVal)
75.
76.     cv2.imshow('HSV', hsvImage)
77.     k = cv2.waitKey(0) & 0xFF
78.
79.     return averageHue, averageSat, averageVal
```

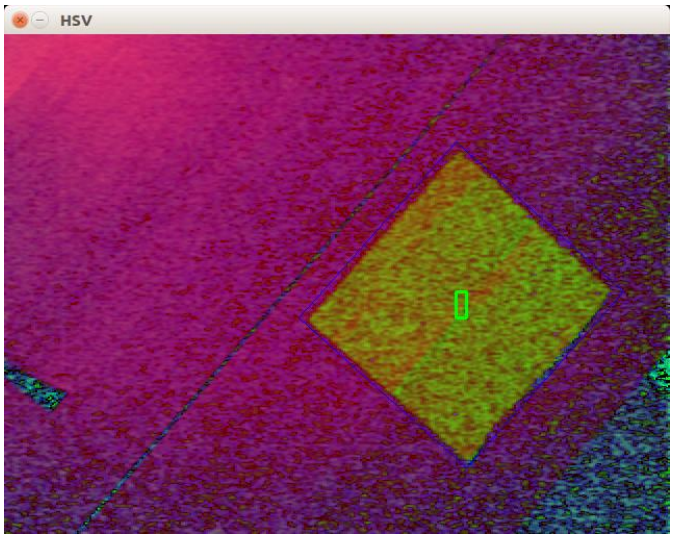
The following images show how the rotation of the target causes the new boxed boundary to change shape. The rotation was simulated by rotating the camera:











With very large rotations the box became very small, so this method was almost useless.

The final method of creating an empty image and drawing on a filled contour that represents the target, means I am able to always look at pixels that are in the same location as the detected target. I have found this method to be far superior than the other methods I have tried, and the algorithm is able to consistently update the HSV values.

1. main.py

main.py is the main file for this coursework, and is the file that should be executed. This file contains the primary loop that will read in a frame from the selected camera, and then do the majority of the image processing.

The algorithm begins by defining a capture device, as I was using my laptop and an external webcam, this is indexed to 1 as 0 is used for the built-in webcam. Parameters of the capturing device are also set so they are consistent every time the algorithm is run.

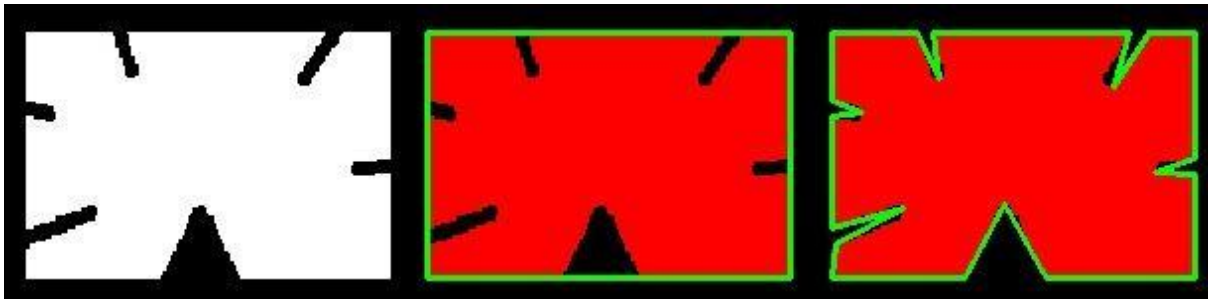
The calibration process then begins, and a script in another file objectDistance.py is run. This returns the focal length of the camera and the known width of the target, and these variables are crucial for calculating the distance of the target from the camera. More on this will be explained later in this document. The next part of the calibration process is creating HSV parameter boundaries. This is done by running a script in hsv_slide.py that allows the user to set the boundaries using trackbars, and see the effect of the selected boundaries on the image. The upper and lower limits are then returned to main.py.

From this point on, the algorithm enters a loop that continually reads frames from the camera and performs image processing.

After the first frame in read, the frame counter is incremented. This variable exists so certain parts of the algorithm can execute at a consistent rate. The inputted frame is then converted from BGR to HSV. This gives a better representation of the pixels based on the light and saturation, and not just the colour. A mask is then applied to the HSV image that's based on the HSV parameter boundaries that were previously set during the calibration process. The resulting image consists of pixels that are within the range of the HSV parameter boundaries.

The algorithm then starts trying to detect objects in the image. This is done through the findContours function, which looks for groups of pixels and creates a list of points that represent the perimeters of the groups. Now that we have a list of contours, we can perform operations that allow us to differentiate between the shapes. The biggest contour is then found, and this is usually the target unless the HSV parameter boundaries have been poorly set.

A for-loop is then used to go over all the contours in the list, looking at them individually. The perimeter of the current contour is calculated, and the shape is approximated. This is done using the approxPolyDP function. This function smooths out the contour into a more simpler shape, and the severity of the smoothing can be changed.



https://docs.opencv.org/3.1.0/dd/d49/tutorial_py_contour_features.html

The second image has a much looser approximation (10%), whereas the third image has a much stronger approximation (1%). As the target will be a rectangle, we do not need such a strong approximation, so I have chosen to go with 10%. The function returns a list of points that represent the corners of the approximated shape. Based on the number of points in this list, we can work out what shape the contour approximates to. If the list has four points, we know the shape is either a square or rectangle, and if the list is any other length, then we can ignore the current contour.

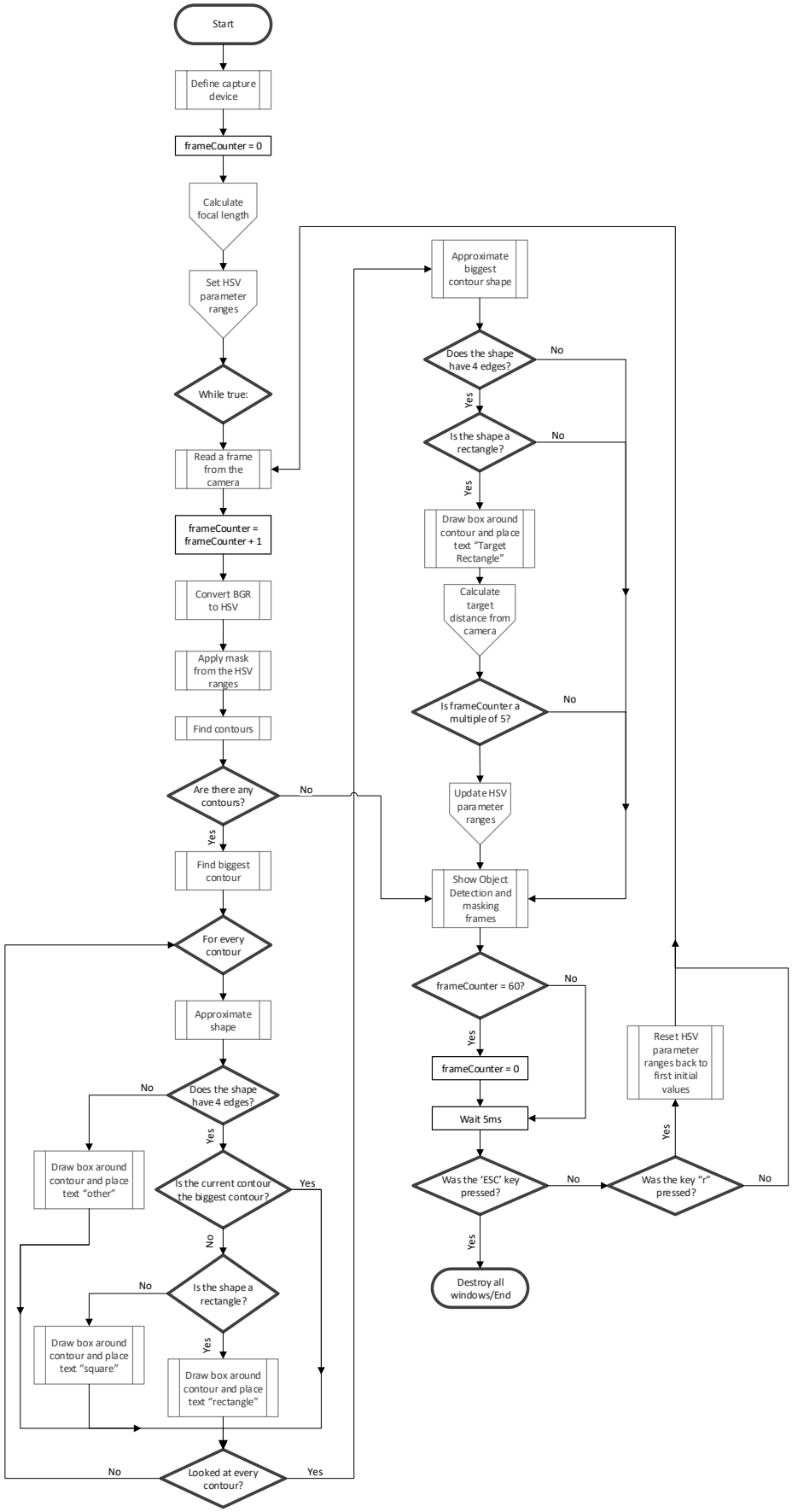
The width and height of the contour can be found from the minAreaRect function. This function returns a rectangle around the contour, and the width and height can be extracted from that. If the width and height are the same, then we know the shape is a square, else it is a rectangle. The algorithm will then place text on the image, labelling the contours with their approximated shape along with boxes around the contours. This allows the user to see what the algorithm is detecting with the current HSV parameter boundaries.

The algorithm then goes back to looking at the biggest contour, approximating its shape and deciding if it is a square or rectangle. If it has detected a rectangle, the text “Target Rectangle” is placed in the middle of the shape. The algorithm then runs the distance_to_camera script in objectDistance.py, and this returns the distance of the target to the camera in centimetres. The distance is then placed in the top left of the image.

The frame counter is then checked to see if it is a multiple of 5. If so, the average HSV parameters are calculated by running a script in update_hsv.py. This script returns the average HSV values of the given target along with new boundary widths. Given the new HSV averages and new boundary widths, the algorithm updates its own HSV parameter boundaries which can then be used to re-apply the mask on the image. This updating is only done if the largest contour is a rectangle, else the algorithm would try to update the HSV parameters based on contours which are not the target.

Once all this is done, two windows will be shown. One of which is the resulting object detection frame, which shows all the detected contours and their approximated shapes. The other shows the resultant masking, so the user can see how the HSV parameter boundaries have affected which part of the frame gets masked.

The algorithm will then wait five milliseconds and see if a key has been pressed. If ‘ESC’ has been pressed, the algorithm will exit, and all windows will be closed. If ‘r’ has been pressed, then the HSV parameter boundaries will be reset back to what they originally were from the calibration process. This is useful if the target has been completely lost by the algorithm, and can be used as a quick reset without having to re-run the calibration process. The algorithm, after checking if a key has been pressed, will then loop back around and read in another frame from the capture device.



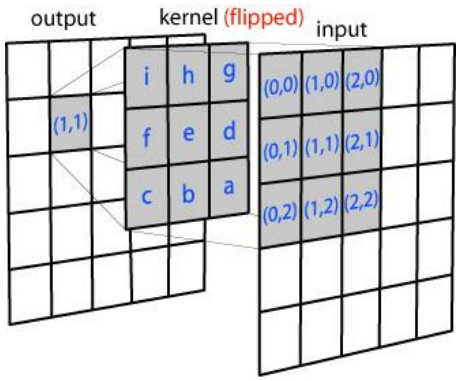
Flowchart for main.py

2. hsv_slide.py

hsv_slide.py is part of the calibration process, and brings up a window with trackbars that allows the user to select the HSV parameter boundaries, as well as view a live feed of how the boundaries affect the masking.

The defined capture device is passed from main.py, and a named window called 'HSV Masking' is created. This is the window that will display the trackbars along with the masked image. The trackbars are then created with set maximum and minimum limits, and placed onto the new named window. After this is done, the script enters a loop that will go on forever unless the 'ESC' key is pressed.

A frame is initially read in from the capture device, and then Gaussian Blur is added. Gaussian Blur is used to reduce noise in an image. This is done by blurring the image using a Gaussian function, where each pixel is multiplied by the Gaussian kernel, and then the values are added up to determine the output for the given pixel.

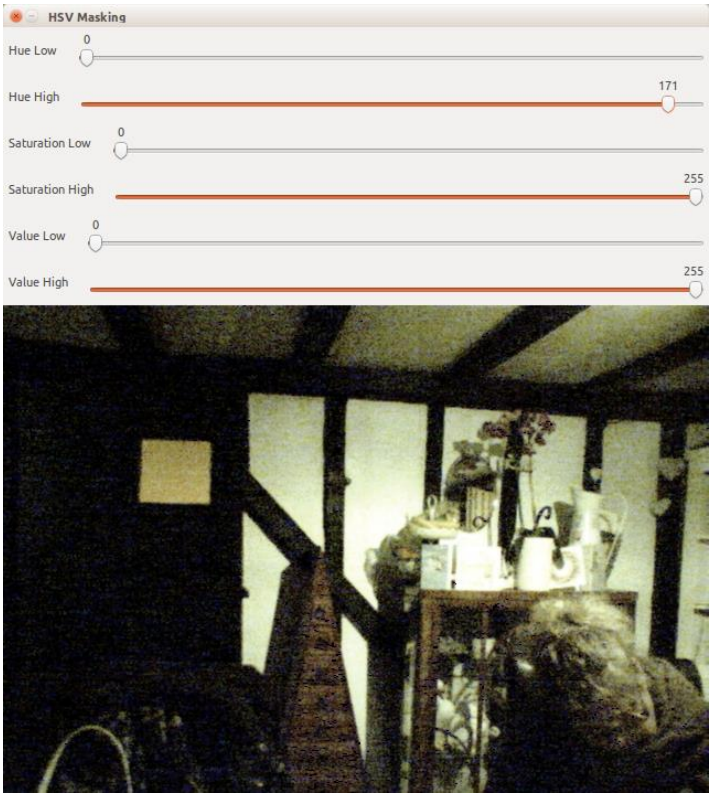


https://i.stack.imgur.com/bRN2c.jpg

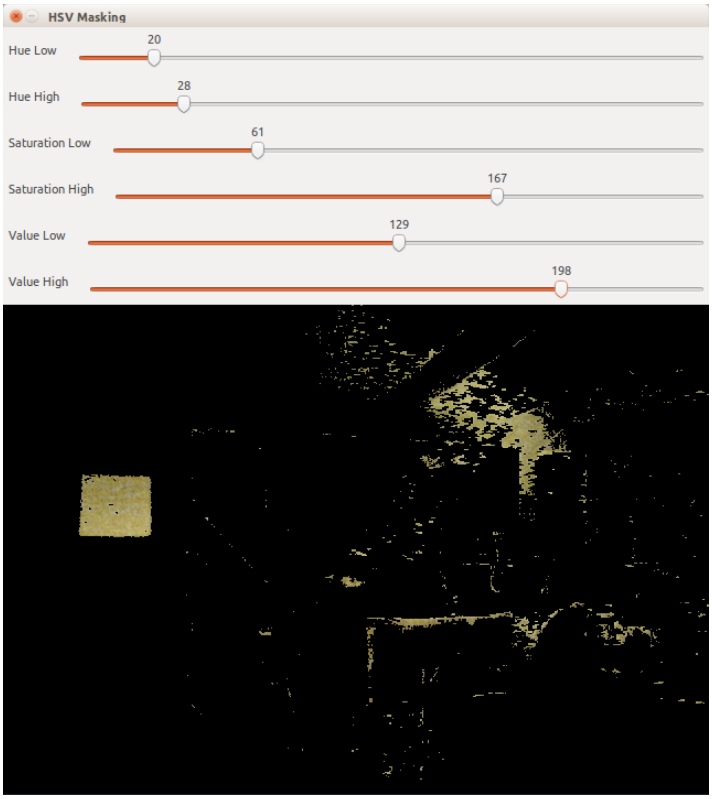
The resulting image has less detail, but is much better suited to applying HSV boundaries as there is significantly less noise in the image.

The colour space gets, again, converted from BGR to HSV. The script will then read the positions of the trackbars and save them as variables. These variables can then be put into two arrays that define the upper and lower limits of the HSV parameters. A mask is then created by applying the upper and lower limits of the HSV parameters on the HSV-converted frame. This mask is an image that only shows pixels that are within the ranged of the HSV parameters. This mask can then be applied to the original frame using the bitwise_and() function, and the resulting image is an image in the BGR colour space, but only shows pixels that are within the HSV parameter ranges if the image was in the HSV colour space.

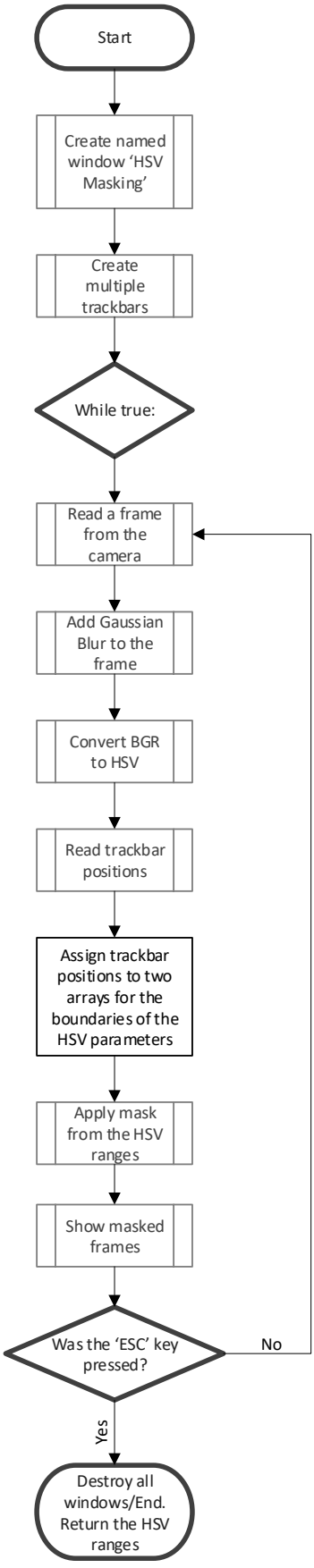
The resulting masked frame is then shown along with the trackbars. The script will then wait five milliseconds to see if the 'ESC' key has been pressed. If not, another frame will be read from the capture device. If it has, the window will close and the two arrays with the upper and lower limits of the HSV parameters will then be passed back to main.py.



HSV limits not set



HSV limits set. The target is clearly shown on the left



Flowchart for hsv_slide.py

3. object_distance.py

object_distance.py contains multiple functions that are used for the calibration process, as well as a function used to determine the distance the target is from the camera. This file was based on the tutorial found at https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/ , which describes how to find the distance of an object using Python and OpenCV, by Adrian Rosebrock.

The main function in this file is runScript(), which is used for calibration. This begins by reading in an image from a file. The file path is pre-defined, so will need to be changed whenever the algorithm is run on a different PC. After the image has been read, the function find_target() is run, which runs a similar algorithm to that which is used in main.py to find the target. The image is converted to HSV colour space, and a mask if applied from pre-defined HSV parameter boundaries. These boundaries have been found by using hsv_slide.py and positioning the camera in the same position that was used to take the image that is being read in.

After the mask has been applied, the largest contour is found, and this should always be the target. A box is drawn around the target and information about the rectangle is returned to runScript(). The perceived width of the target is then found by picking out part of the information returned by find_target(), and this is a value in pixels. Now the perceived width is known, the focal length formula can be applied.

Focal Length = (Perceived Width * Known Distance) / Known Width

Known distance and known width are already defined as the width of the target has been measured, and the distance the target was from the camera when the image was taken was also measured. Now the focal length is known, the formula can be rearranged to calculate the distance the target is from the camera.

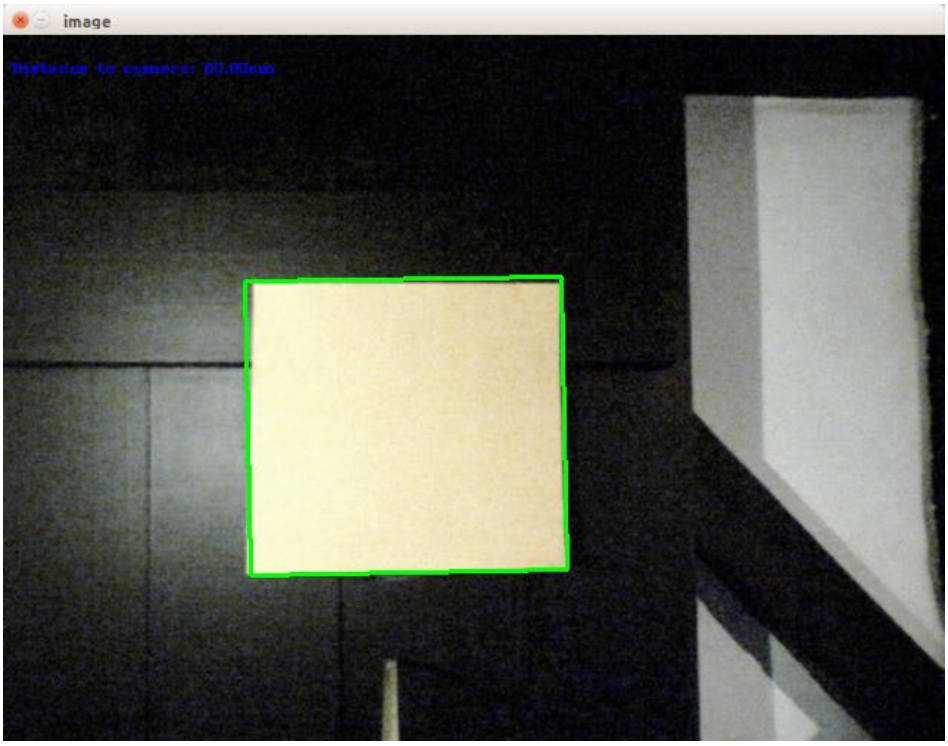
Distance from camera = (Known Width * Focal Length) / Perceived Width

This calculation is done in a separate function called distance_to_camera(), and this can be called from main.py once a target is found. The function returns a distance in centimetres. runScript() then continues and draws the found box contour onto the image, as well as placing text in the corner saying “Distance to camera: %.2fcm”, with the distance being parsed to the placeholder. Finally, the image is shown so the user can see how well the contour is drawn around the target. This is crucial because if the box doesn’t match up to the target very well, the calibration data will be off. The calculated focal length will be incorrect and this will cause the distance measurements to be off when distance_to_camera() is run from main.py. The function then waits indefinitely for any key to be pressed.

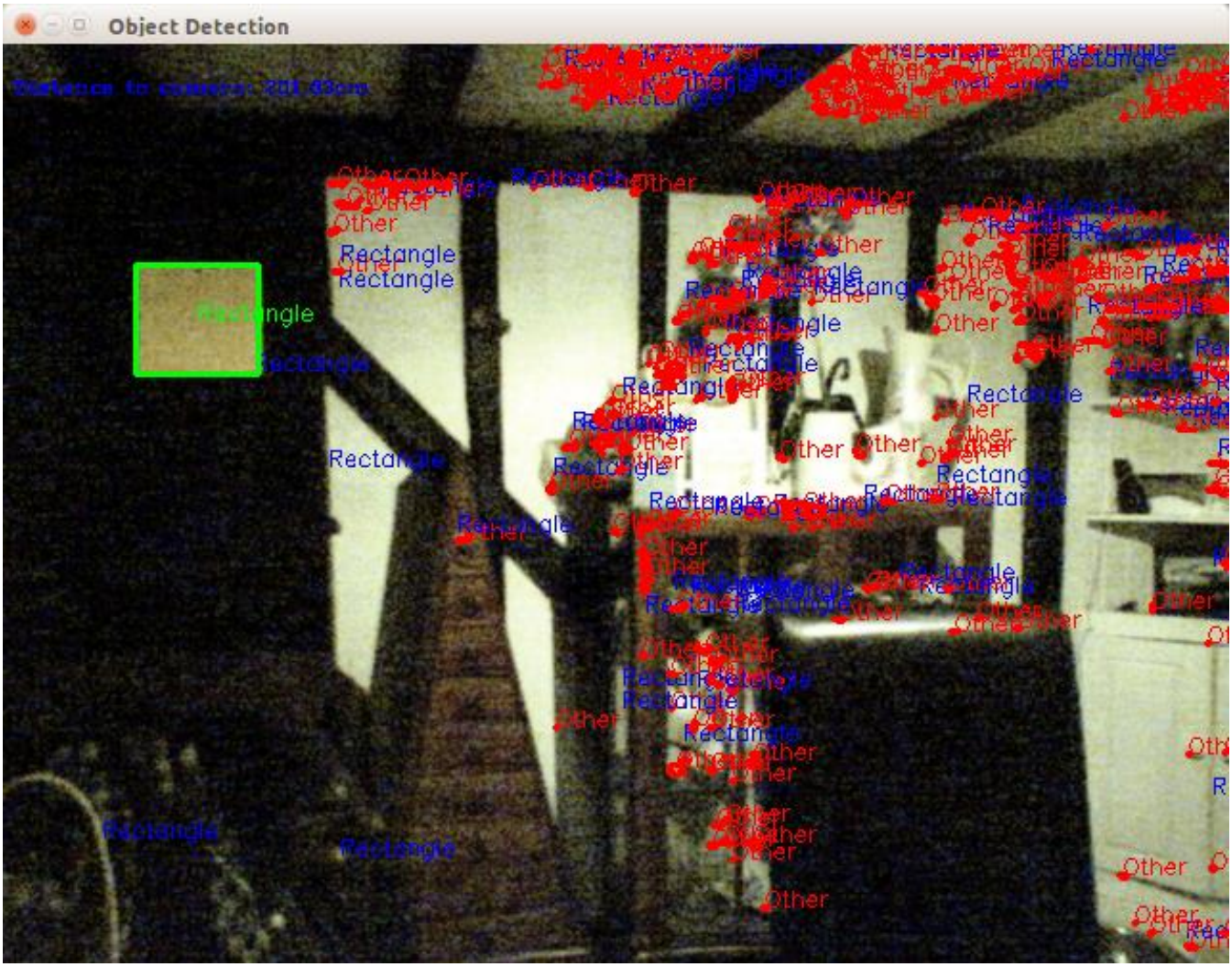
I had originally started by placing the camera 30cm away from the target and taking a snapshot, but being very close caused the target to not be square on, so the box contour that was drawn onto the image did not line up with the target. This would cause all future distance measurements to be off. I then decided to take a snapshot 60cm away from the target. This gave a much more accurate box contour around the target, so my calibration was reasonably accurate. The calibration process is run every time main.py is run, and because the algorithm uses the same training image every time to calculate the focal length of the camera, then the calibration is consistent. This does, however, mean that if another capture device was used, then a new snapshot would need to be taken for the

calibration process, so my current method limits the transferability of the code. The algorithm is also limited to using whichever target was used in the calibration process, and if a different target is used that has a different width, then the distance calculation will be wrong.

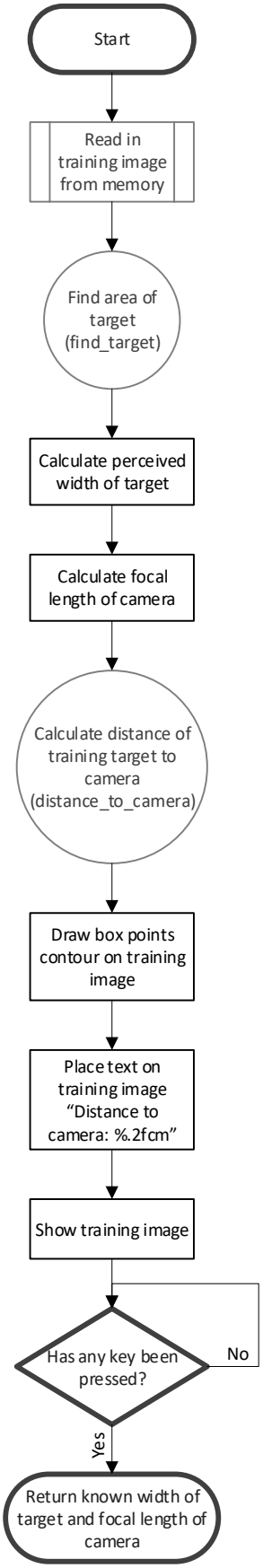
Calibration image, 30cm from camera



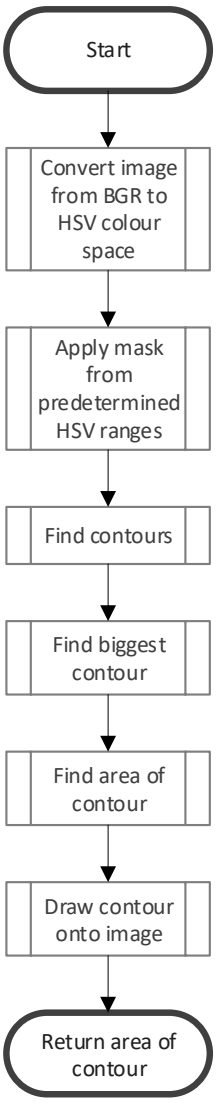
Calibration image, 60cm from camera



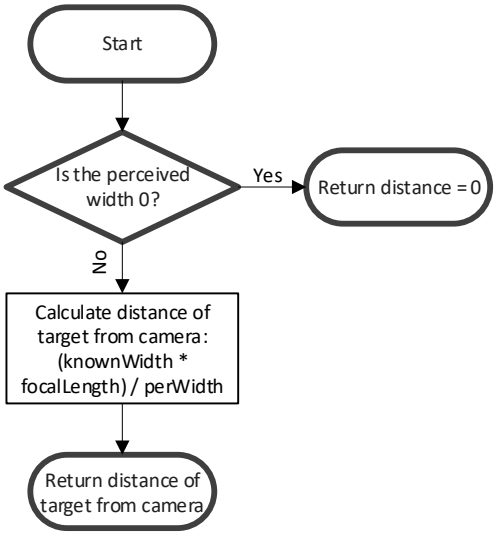
main.py being run with distance calculation. Calculated distance = 201.63cm. Measured distance = 199cm. Algorithm detects a lot of other contours due to target being a similar colour to the rest of the image



Flowchart of runScript() from object_distance.py



Flowchart of find_target() from object_distance.py



Flowchart of distance_from_camera() from object_distance.py

4. update_hsv.py

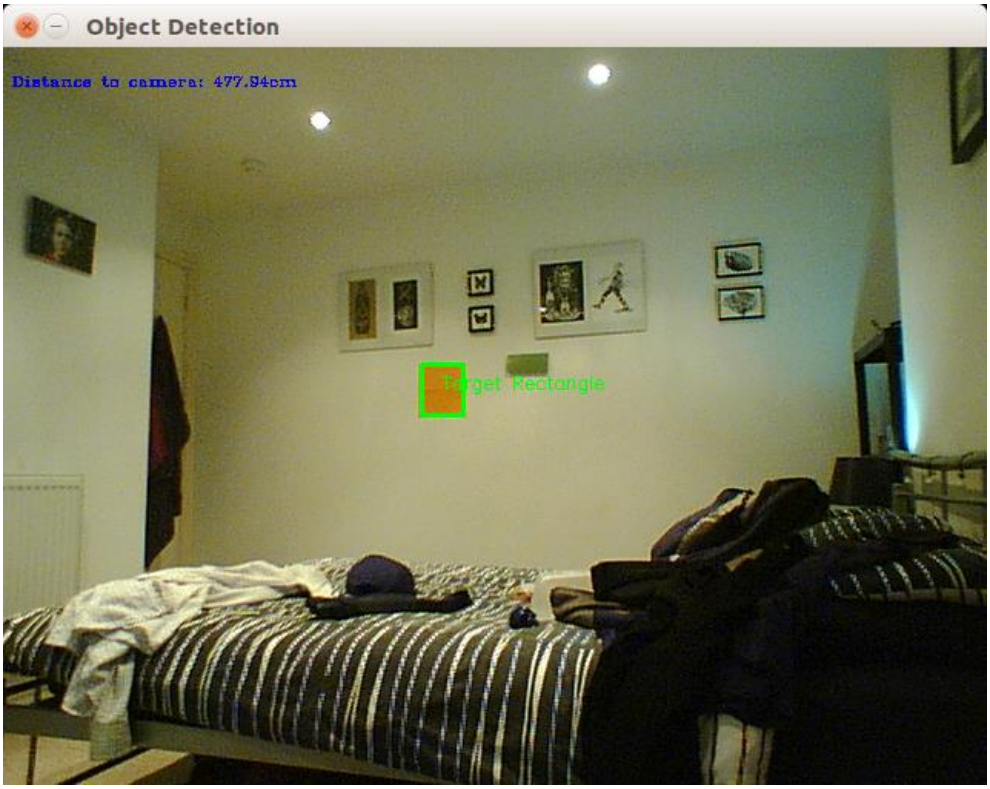
To improve the robustness of my application, I have decided to dynamically update the HSV parameter boundaries. This means that any slight alterations in light should not affect how well the algorithm can detect the target, as the parameters will update and adjust.

The algorithm works by reading in a HSV frame parsed from main.py and applying Gaussian Blur to it. A blank image is then created with all black pixels. The box contour from the detected target is then drawn onto the image. This contour is not just an outline, but is a solid shape that matches the target. The shape is a uniform colour, so now the algorithm searches for pixels that match the set colour of the shape, and returns a list of X and Y coordinates of those pixels. Now the algorithm has a list of X and Y coordinates, it can use those coordinates to look at pixels in those locations from the HSV frame.

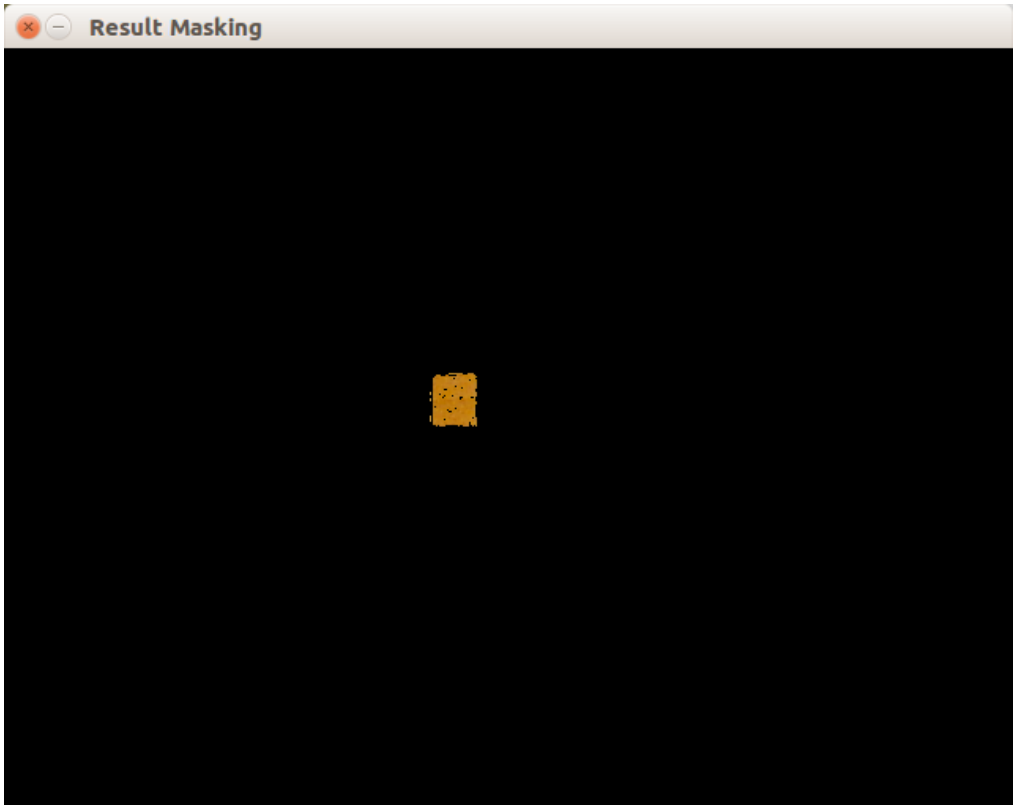
The algorithm then creates another list, this time of the HSV intensities of all the pixels from previous coordinates list. It then loops over this list, adding up the total hue, saturation and value of all pixels, and finds the maximum and minimum of all HSV parameters. After this, the average hue, saturation and value are found. It will also find out the ranges of each parameter. This needs to be known so we know how much above and below the average HSV parameters we need to set the new boundaries. I decided to use a boundary width of 33% of the range. If I kept the boundary width to be 100% of the range, then the algorithm would start to find other pixels in the image that were within the boundaries, and sometimes detected larger contours that weren't the target, throwing the detection off.

After these calculations are done, the empty image is shown to the user and the average HSV values, along with the new boundary widths, are returned to main.py. Updating the HSV parameters only occurs if the largest contour is a rectangle and the frame counter is a multiple of five. This stops the algorithm trying to update the HSV parameters on parts of the image that aren't the target.

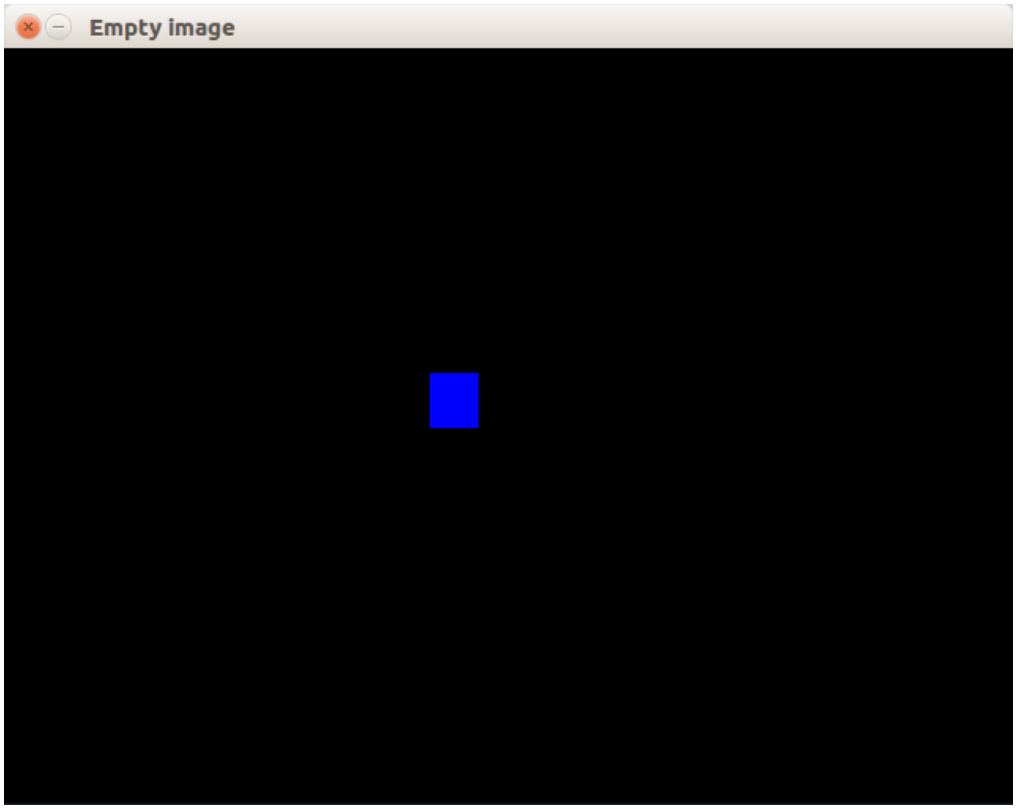
This robustness solution will only work if there are slow changes in light on the target. If the illumination of the target decreases rapidly, the algorithm will not be able to update the HSV parameters as the change in light will cause it to lose the target. If no target is found, then the script in update_hsv.py cannot run.



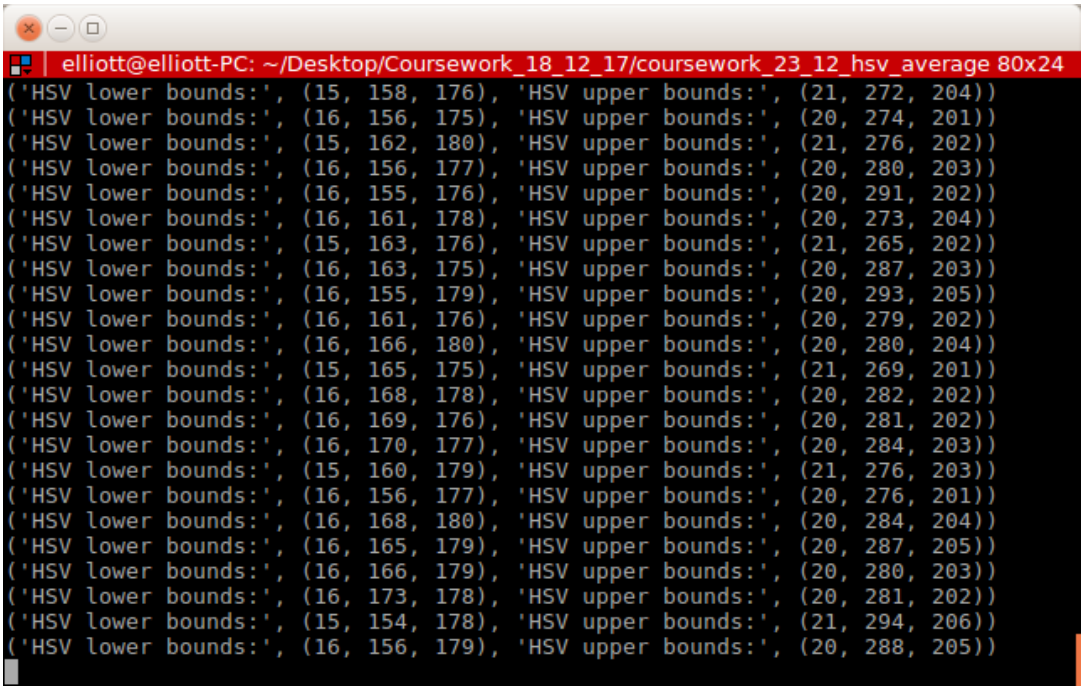
Final output window



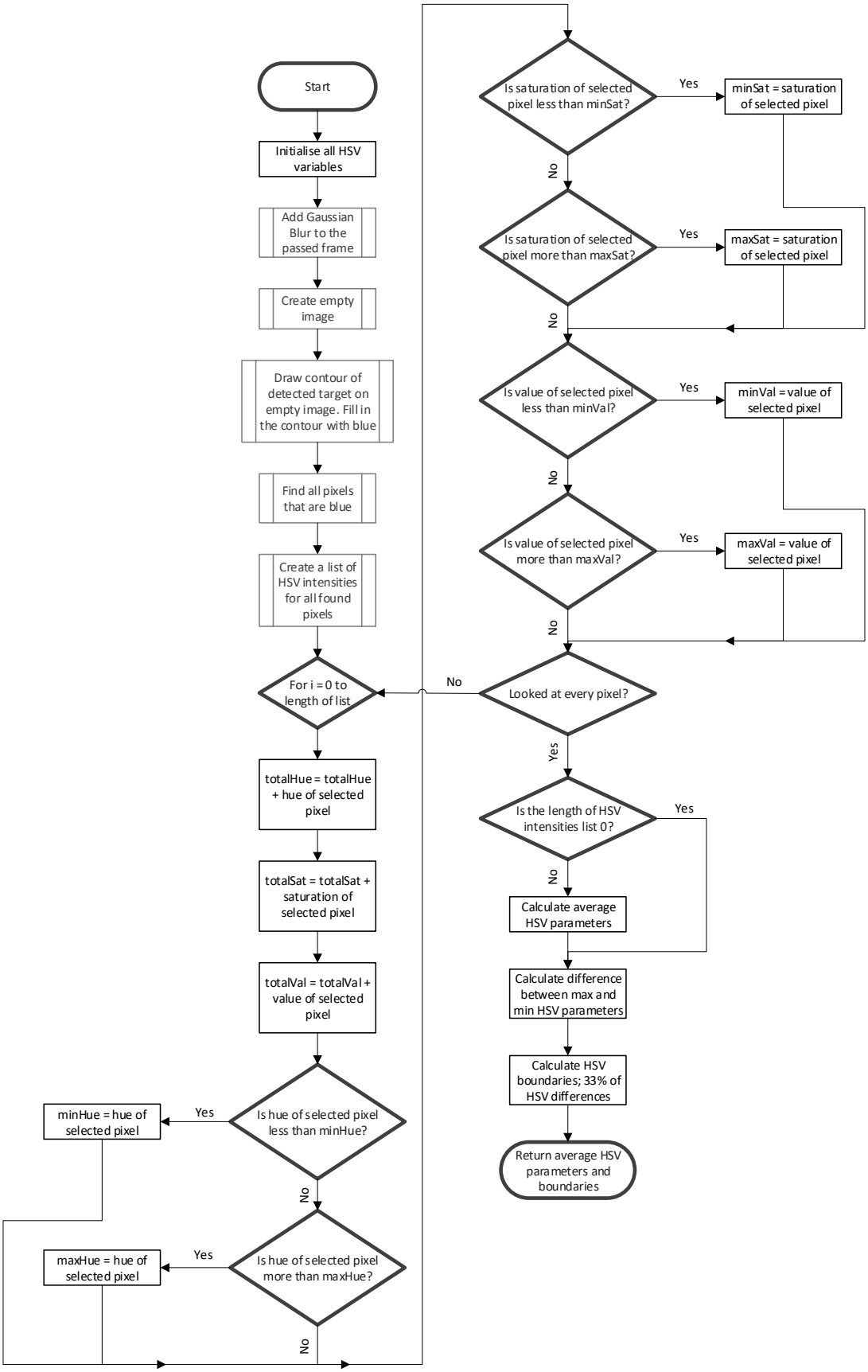
Masked image, showing the pixels that are in the HSV boundary ranges



Empty image with the filled in contour shape



Terminal output showing the HSV boundary updates



Flowchart for update_hsv.py

How Specific Is My Solution?

Many aspects of my solution can be considered as very specific. The algorithm will only regard rectangles as being the desired target. Squares and other shapes can be found, but the algorithm will deem them as un-useful information. For the distance calibration, the algorithm needs an initial image to calculate the focal length of the camera, and it also needs to know the actual width of the target. This means my solution will only work with one target: that which is used in the calibration process. If any other target is used, then the distance calculations will be incorrect. If another target is to be used, then a snapshot of the target at a known distance away from the camera will need to be taken and implemented into the code. The width of this new target will also need to be incorporated into the algorithm.

The colour of the target is also quite restricted. The target will need to be a colour that is of a high contrast to its surroundings, otherwise the algorithm may detect larger contours that fall within the HSV parameter boundaries and not detect the target. The target, therefore, needs to be the largest detected contour in the image at all times, otherwise the detection will fail.

My solution can be used for different target detection, but modifications to the code will be necessary. Different calibration images and changes to how many sides the desired target has will need to be implemented for different-shaped targets.

Discussing the Concept of Identification

For a robot to fully interact with the world, it must process its surroundings. This can be done through a range of sensors, with the most complex one being cameras. Object recognition and identification is no simple task, and is something that we take for granted. Lower animals, such as insects, can have uncomplicated vision systems that result in complex behaviours. An organism's response to light with motion is known as phototaxis, and positively phototactic creatures, such as moths, move towards light sources. On the other hand, negatively phototactic creatures, such as woodlice, will move away from light. These complex behaviours from basic sensors allow the organisms to survive.

Theory of Recognition

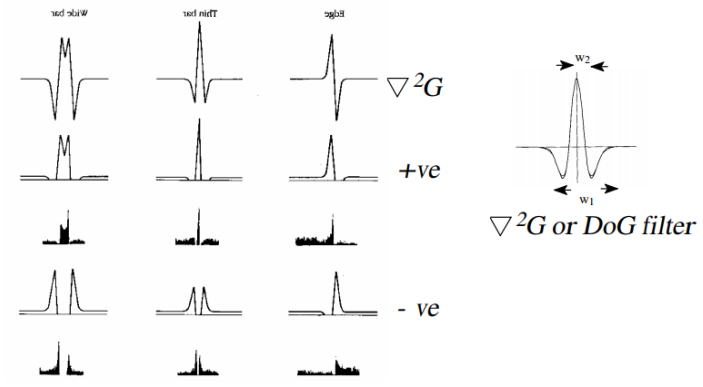
All models of recognition require three primary characteristics:

- Rotational invariance. People are able to recognise objects from many different angles, as well as when the objects are partially obscured[i].
- Size invariance. Objects can be recognised when scaled up or down.
- Translational invariance. Regardless of where the object is in the field of view, it can still be recognised.

The combination of these three characteristics creates the problem of 'pose', where an unknown object can be in any possible position, yet the observer is still expected to recognise it. Many other factors increase the difficulty of recognising objects, and these can include an object's opacity, flexibility and whether there are multiple parts joined together.

The ganglion cells in the eyes function as convolution processors. They use a Gaussian kernel which varies according to the position of the pair's receptive field. They are positioned into a central disk and a concentric ring, with both regions responding oppositely to light. The pair of cells consist of two types; off-centre and on-centre. Each implement a convolution with a Gaussian with, off-centre behaving oppositely to on-centre. The becomes a Difference of two Gaussian receptive fields (DoG). Retinal ganglion cell receptive fields give information about discontinuities in the distribution of light falling on the eye, and this can specify the edges of objects.

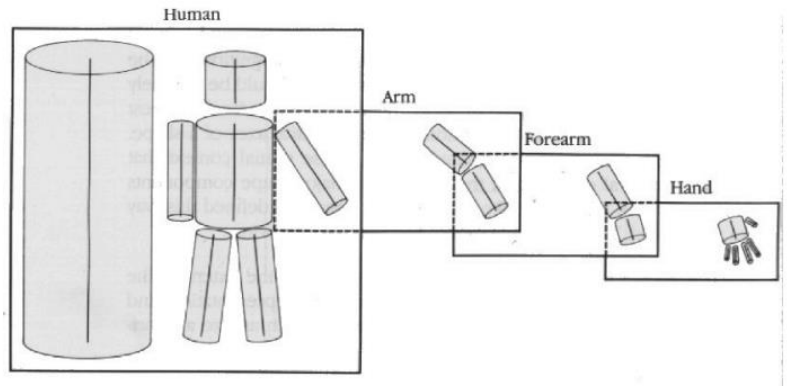
Gangion cell responses to an edge, 1° (0.5 on- centre) and 5° (2.5 on- centre) bars



adapted from Marr, D. (1982), Vision, p.66.

Second order derivative of Lapacian $= d^2/dx^2 + d^2/dy^2$
ie. zero- crossingsare centre points of contrast edges

Marr and Nishihara developed a computational model of vision that had three stages, also known as sketches. The first, or primal raw sketch, performed local clustering of edgelets, blobs and zero-crossings. The second, or two and a half dimensional sketch, grouped surfaces together and made explicit boundaries. The surface shape is extracted from stereoscopic vison, shading and motion. The last, or three-dimensional sketch, matches surfaces with generalised cones, and then provided a database of objects based upon cone relations. An object model exists at a range of detail, from coarse, where one vertical cone matches a human, to fine, where a set of cones matches hands and fingers[ii].



There are problems with this model of visual perception. Although the evidence for the primal sketch is strong, the support for the higher order processing is sparse, and in the case of the 3D sketch, is non-existent. Much of the two and a half dimensional sketch, although supported by computational algorithms, has little evidence from animal studies.

Kirkpatrick explores the value of the Geon theory of visual recognition[iii], that proposes recognition by components. The theory states that objects are made up of Geons (geometric ions) that, when combined, can compose all objects. Biederman calculated that a minimum of 36 Geons are needed, and extended to 42 for full 3D modelling. Biederman proposed non-accidental properties of image features:

- Co-linearity: straight lines in images are caused by straight lines in the world
- Curve-linearity: smooth curves in images are caused by smooth curves in the world
- Symmetry: Symmetrical regions in images arise from symmetrical objects
- Parallelism: parallel lines in images arise from parallel edges in the world
- Co-termination: lines terminating together in images arise from edges terminating together in the world

Geons are based on 2D non-accidental properties, so they can be detected in images without prior processing. Clustering Geons into objects may be sufficient for future recognition, but if the object if obscured by other things, then the task of matching the Geon cluster to an object is more difficult.

Machine Vision

The task of recognising objects can be broken down into:

- Extracting features
- Group features
- Associate groups with an object
- Verify objects

The difficulty of detecting objects within an image is proportional to the environment in which the image is taken. If a high-grade industrial camera with controlled lighting is used to identify missing objects on an assembly line, then the system will perform well as the variation in pose is very limited.

Edges can be easy to detect, but as soon as an image gets noisy, the task of detecting edges can increase in complexity. Edge detection works by looking for sharp differences in pixel brightness, or has discontinuities. In the perfect situation, running an edge detector algorithm on an image will lead to a set of connected curves that give the boundaries of objects. This can reduce the amount of information in an image to get rid of data that may not be relevant.

Regions of similarity can also be easily detected. Similar colour, grey level and texture can all be looked for in an image, and grouped together. The algorithm will need upper and lower thresholds to allow it to group pixels within the ranges. The resulting image will consist of pixels that are within the thresholds, and will resemble small islands. This will give the algorithm explicit regions, but not objects. Split and Merge by Horowitz & Pavlidis[iv] improves the clustering by performing local thresholding to identify the pixels that need to be grouped.

Shapes can be analysed by observing the rotations of edges of an edge-detected object. A boundary can be described in many ways; Snakes (Staib & Duncan[v]) and B-spline curve fitting (Liang et al[vi]). Once fitted, the shapes and curves and be used to recognise objects.

Combining all these methods allows computers to be able to detect shapes and objects in images, and progressing the ability of object recognition relies on understanding natural images, and using studies of natural vision systems to develop more accurate algorithms.

---

[i] Biederman I & Gerhardstein PC (1993) Recognizing depth-rotated objects: Evidence and conditions for three-dimensional viewpoint invariance. Journal of Experimental Psychology: Human Perception and Performance, 19, 1162- 1182.

[ii] Marr D and Nishihara HK (1978) Visual information processing: artificial intelligence and the sensorium of sight, Morgan Kaufmann Readings Series. ISBN:0-934613-33-8

[iii] Biederman I (1987) Recognition-by-components: A theory of human image understanding. Psychological Review, 94, 115-147

[iv] Horowitz & Pavlidis, 1974, Picture segmentation by a directed split- andmerge procedure, Proc. 2nd IJCPR, p.424 433.

[v] Staib LH, and Duncan JS (1992) Boundary Finding with Parametrically Deformable Models, PAMI(14), No. 11, November 1992, pp. 1061-1075

[vi] Liang KM, Mandava R & Khoo BE (2003) NURBS: A new shape descriptor for shape-based image retrieval. International Conference on Robotic, Vision, Information and Signal Processing, Penang, Malaysia, 141-146.