4

Color Classification and Object Recognition for Robot Soccer Under Variable Illumination

Nathan Lovell and Vladimir Estivill-Castro
Griffith University
Australia

1. Introduction

One of the biggest challenges for vision systems in mobile, autonomous robotics is to show adaptivity to changing visual circumstances. Many state of the art robotic platforms still use color segmentation as the basis for vision, even though this technique performs badly when the robot is removed from the exact lighting environment for which it was calibrated. Nevertheless, the very nature of a mobile platform with cameras on board, suggests that it will move and thus lighting conditions will change.
All variants of human soccer regularly use color-coding, but robotic soccer is currently limited to soccer fields with carefully controlled illumination conditions. Experiments in the RoboCup competition have shown how detrimental even the smallest lighting changes can be to platforms with cameras on board. Competitors calibrate their systems multiple times, even on the same field, simply because the ambient light in the room varies with the time of day. Small variations adversely affect the ability of agents to accurately assess the visual scene. Robotic soccer requires accurate visual information and our current systems simply do not provide this under variable or unknown illumination conditions.
Many attempts to resolve this issue involve a priori geometric information regarding the objects in the visual scene. For example, if I know that the only round thing that I expect to see is the ball, then I can identify the ball by its round shape —I do not need its color. Such techniques are useful but often require far more intensive computation. Robotic soccer is a real time adversarial, nondeterministic and inaccessible setting, on a platform with limited computational power. Therefore, some geometric image processing solutions are simply not suitable.
In this chapter, we describe our solution to this problem. Our solution is fast enough to run on the limited resources of a mobile system under the constraints of real-time image processing. Our technique is still basically a color segmentation process, so it runs in equivalent time to systems that are already used for robotic soccer. Faster processors or more reliable hardware (cameras/lenses) will result in more reliable and more robust systems under even larger illumination variations.
We do not attempt to exhaustively classify an entire color class for a single illumination condition. Rather we calibrate a core color class of, say, orange, that will remain orange under many illumination conditions. This sparse classification means that in no image will

Source: Robotic Soccer, Book edited by: Pedro Lima, ISBN 978-3-902613-21-9,
pp. 598, December 2007, Itech Education and Publishing, Vienna, Austria

every pixel that looks orange be classified as orange, however in every image some pixel that looks orange will be classified as orange. We can quickly find some pixels of any given class, independent of the illumination condition, and complemented with other techniques, we are able to identify objects.
Simple geometric shapes are usually easy to locate. A circle, for example, requires three edge points. We can thus use a special edge-detection algorithm to quickly find three points on the edge of the circle; starting from any of the pixels we have identified. A more complex shape may require a description of the entire edge. For this purpose we have developed a fast edge detection algorithm that can be used in conjunction with a border following algorithm so that we do not have to process all of the edges in the entire image.
A large proportion of the machine vision literature of recent times, both in RoboCup and elsewhere (Gunnarsson et al., 2005; Shimizu et al., 2005; Kak & DeSouza, 2002), has attempted to address the problem of dynamic illumination conditions. We present a viable solution here, provided that the conditions are not too widely variable. Our technique has been applied with success to the Four-Legged league in RoboCup. We have illustrated our ability to correctly identify objects on the field in real time in complex and dynamic lighting environments. The object recognition system is efficient for the RoboCup environment and is robust to changes in color intensity and temperature. It is based on a sparse classifier that is very accurate, but only on pixels that are at the core of each color class. It essentially refuses to make a decision for most other pixels, labeling them as unknown. The necessary ideas of image segmentation appear in Section 3 and the use of the sparse classifier appears in Subsection 5.1. Optimized edge-detection is the second pillar of our approach. Thus, Section 4 shows how we find the border of objects without performing complete edge detection. How everything comes together is described in Section 5. Detecting object boundaries in simple objects is discussed in Section 6, while Section 7 handles more complex objects. Accuracy of our methods is reported in Section 8. We offer final remarks in Section 9.

2. Pipeline for Color-Coded Environments

A vision pipeline is a sequence of techniques and algorithms applied in a pipeline architecture (Shaw & Garlan, 1996), where each step in the pipeline manipulates or analyses

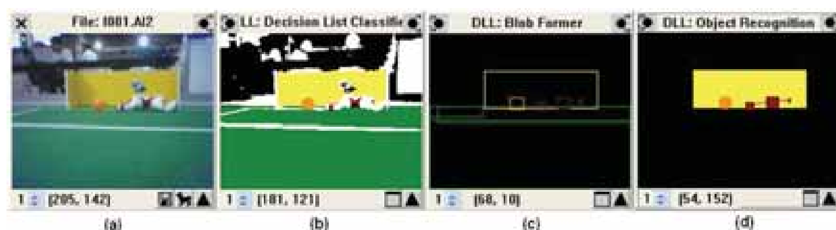

Fig. 1. The Bruce et al image processing pipeline

image data. There are certainly many varied methodologies for image-processing solutions and it is also true that robotic vision systems reflect this variety depending on the specific application context of the system. However, across a variety of problem domains within mobile autonomous robotics, a similar pipeline has emerged as a standard not only as a

standard in RoboCup (Ogihara et al., 2005; Veloso et al., 2005; Chalup et al., 2004; Chen et al., 2003), but also in other robotics applications (Halme et al., 2000; Kak & DeSouza, 2002).
This pipeline was first described by Bruce, Balch and Veloso (Bruce et al., 2000) in the context of the RoboCup competition and is shown in Fig. 1. The first stage in the pipeline is image segmentation (b) where each pixel in the image is labeled as one of a set of color classes. Pixels that look, for example, blue, are labeled as belonging to the class blue. After the image is segmented, it is passed to a blob-former (c). Blobs are groupings of connected pixels that all belong to the same color class. Each blob can then be analyzed to determine its properties or relation to other blobs. This is the object-recognition stage (d). The Bruce et al. pipeline is certainly not the only pipeline used by mobile, autonomous robots. Indeed, even within the RoboCup competition we have seen many interesting and significant variations in recent years, such as the one in the German team code (Röfer et al., 2005). However, the

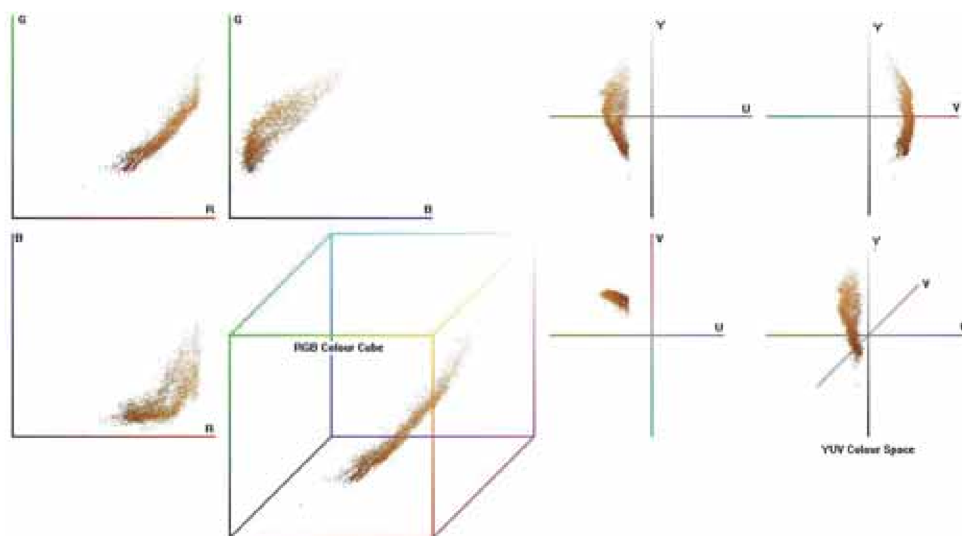

Fig. 2. All the (orange) ball pixels in Fig. 3 mapped to the RGB colur spac (left) and the YUV-colour space (right). Note that the shape in space defined as orange is not regular

Bruce et al. pipeline does possess certain advantages that make it an extremely popular choice. Firstly, it is very efficient — it requires only one pass over the raw data and one pass over color segmented data in order to complete the object recognition task. It is relatively easy to implement and there are third-party libraries available that implement some of its functionality. Another advantage is that it is also relatively easy to calibrate and to use in various different conditions. So significant are these advantages that even some competitive teams do not deviate far from the pipeline, despite its age (Chalup et al., 2004; Chen et al., 2003).
In the context of real-time autonomous robotic vision, color is one of the only object features that is sufficiently easy to distinguish in order to make the image processing fast enough to keep up with the frame rate of the camera. A pipeline similar to Bruce et al. can even be

used in military hardware in order for missiles to detect targets via the infrared spectrum (Shaik & Iftekharuddin, 2003).

3. Colour Classification

We describe a color classifier tailored for the first part of the pipeline, namely, image segmentation. The classifier described here is as fast as a lookup table, but considerably more compact than any other classifier available (it is under 4KB). Also, in contrast to other classifiers used at RoboCup, its representation is intuitive.
We may represent a color classifier as a function

$$\texttt{colour_class}: Y{\times}U{\times}V \rightarrow \text{colors} \quad (1)$$

that given a triplet (y,u,v) produces a color class (a member of a discreet set of color classifications). This function may be easily represented as a single characteristic function for each member of colors. For example, let class_orange: Y×U×V→ {true, false} return true when the pixel (y,u,v) belongs in the class orange.
The set of all pixels in the color space that we would like to classify orange cannot be accurately described by any linear discriminator (refer to Fig. 2). This is because the orange area is not regular. This makes the individual characteristic functions difficult to define.

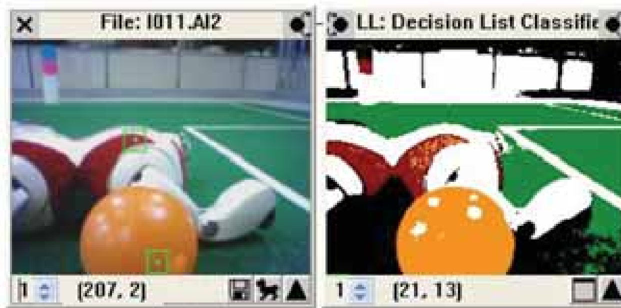

Fig. 3. The highlighted pixels both classify to orange because they have the same component (YUV) values. But the human eye clearly perceives one as red and the other as orange because of the context of the image

It is clear that a more complex knowledge-representation alternative would be required to exactly represent each color class. We certainly lose accuracy whenever we attempt to define the class orange by any linear discriminator. We are, however, quite unconcerned with this loss of accuracy. The reason why such accuracy is unimportant is illustrated in Fig. 3. The two highlighted pixels in the image on the left have exactly the same (*y,u,v*) tuple even though our eye clearly perceives one as red and one as orange due to the context of the image. Should that pixel be classified as orange or red? Clearly the human eye, to some extent, defines color by what it expects to see given the surrounding context of the image. An ideal classifier would therefore classify a pixel as class orange only when the predominant color of the surrounding shape is orange. The problem, of course, is that image

segmentation is usually the first step in object recognition (and such recursion could be CPU-costly). The first image analysis algorithm does not yet know the context of the object in which the pixel lies. For this reason, even a comprehensive look-up table (an artificial neural network, a support vector machine, or a decision tree), will classify pixels incorrectly. For this reason, we will be content with a classifier that recognizes the core class, where core means those values (y,u,v) that remains on the same class across a wide variety of illumination conditions. We will ignore pixels that could fall into more than one class depending on surrounding context.
Thus a composition of linear descriptions will be a suitable representation. We start with some simple characteristic functions. A characteristic function of each color class can therefore be represented by the projection functions on each of the dimensions Y, U and V. For example, the three projections Y_{orange}, U_{orange} and V_{orange} can be used to approximate `class_orange` in the following way:

$$\texttt{class_orange}\,(y,u,v) = Y_{orange}(y) \wedge U_{orange}(u) \wedge V_{orange}(v) \tag{2}$$

where $\wedge$ stands for logical AND.
Each of the characteristic projection functions has a domain of 256 values and thus can be feasibly stored in a look-up array (of size 256 bits) that stores 1 if Y_{orange}, is true and 0 otherwise. Thus, there is a look-up table of 256 values for each of Y_{orange}, U_{orange} and V_{orange} which we store in 3 arrays. A characteristic `color_class` function can then be represented by a C++ bitwise AND-operation:

$$\texttt{color_class}\,(y,u,v) = \texttt{Y[y]\&U[u]\&V[v]}. \tag{3}$$

We can store the look-up for several characteristic functions in compact arrays of `int` type (rather than Boolean type) if they are of convenient width (32 bits is a convenient width because it represents an `int` data type on a modern 32-bit processor). We do this by putting the look-up for the first characteristic function in the first bit (left-most bit) of the value, the second function look-up in the second bit, and so on. If more characteristic functions are required, then we simply increase the size of the data type.

Algorithm 1 Decision List classification.

Require: Arrays `Y [255]`, `U[255]` and `V [255]` where the first bit in `Y[n]`, `U[n]` and `V[n]` is the look-up of the characteristic function for the n-th color class. The color tuple to classify is (y, u, v).
Ensure: The color class of (y, u, v).

```
1: val = Y[y] & U[u] & V[v]
2: count = 0
3: while !(val&0x01) && (count < 32) do
4: val=val>>1
5: count++
6: end while
```

However, there are typically few color classes. Thus, we can use more characteristic functions and consider them organized into a hierarchy. The left-most bit represents the highest decision rule. If a pixel is not already classified, we proceed to the next characteristic function, until one rule classifies it. The Machine Learning or Data Mining community would refer to this as a Decision List (Witten & Frank, 2000). This can be implemented with efficient shift operations that compute the discrete $\log_2$ of the result to determine the ID of the color class. This entire method of classification is shown in Algorithm 1.

This implementation of our algorithm is similar to that presented by Bruce et al. (Bruce et al., 2000). The innovation of our approach is that each color class may be represented in the array more than once because Algorithm 1 retrieves an index to the class, not the class itself. There are two advantages to this system over Bruce et al.. The first is that our algorithm allows us to discriminate non-rectangular regions in the color space using a decision list format. The second is that the representation of the calibration file is very easy to understand and edit (even by hand). This is advantageous when learning a classifier, validating the classifier, or inspecting the pipeline functionality with tools that link to the AIBO. The Bruce et al. algorithm does not permit more than one linear discriminator for each color class, and therefore does not permit non-rectangular color regions. This makes the Bruce et al. algorithm unsuitable for use in both of the calibration techniques (robust and sparse) that we will introduce shortly.

Algorithm	Average time on AIBo per frame (ms)	Amount of memory consumed (bytes)
Our method	1.71	3060
Complete look-up table	1.41	16777216 (16Mb)
k-Nearest Neighbours	Depends on k but very slow (1ms / k)	fairly small (4 bytes .k)
Support vector machines	1.93	262144 (256Kb compressed)

Table 1. Comparison of our Decision List classifier with other available methods

Our entire classifier is at most 3060 bytes in memory and runs very quickly. Table 1 compares our method with some of the other classifiers that are being used in the RoboCup competition. It is easy to understand why our technique is so much faster than the others. If classification is treated as a spatial problem (k-Nearest Neighbors), then the classifier is required to compute Euclidean distances that involve a square root operation. If it is treated as a decision- tree, then it may process up to 20 levels of conditional statements before a decision is reached. Our method has a runtime cost only marginally larger than the fastest possible solution of the look-up table.

In general, calibration for classification is a supervised learning task; given a set of sample pixels with known color class, derive a classifier to assign a color class to future pixel values. It is important to recognize that we may not encounter every possible (y, u, v) tuple in the training set, so the classifier must generalize. We consider two types of calibrations possible: *robust* and *sparse*. Let P be a training set of n images and $Orange_i$ be the set of pixels that we wish to classify as orange (for example) within the i-th image. Then an ideal robust classification for the class orange is:

$$\texttt{class_orange}\,(y,u,v) = \text{true} \Leftrightarrow (y,u,v) \in \bigcup_{i=1}^{n} Orange_i \,. \tag{4}$$

That is, if a pixel with values (y ,u, v) is recognized as orange in *any* of the images in the training set, then the classifier should label this pixel as class orange for any future images. Of course, an ideal robust classifier may not be possible (the same (y, u, v) tuple may be assigned to two different classes in the training set even within the context of the same image). An ideal classifier may also suffer from over-fitting (Witten & Frank, 2000). In practice, we weaken the condition by asserting that the classifier should label a pixel as class orange if it is recognized as orange *more often* than any other color class.
By contrast, an ideal sparse classification for the class orange is:

$$\texttt{class_orange}\,(y,u,v) = \text{true} \Leftrightarrow (y,u,v) \in \bigcap_{i=1}^{n} Orange_i \,. \tag{5}$$

That is, we label a pixel (y, u, v) as orange only if it is recognized as orange in *all* of the images in the training set. Again, sometimes it is necessary to weaken this condition. In practice we label a pixel as orange, if it is recognized as orange in *most* of the images in the training set.
There are both benefits and drawbacks to each of these two calibration methods. Because we are aiming for versatility to illumination conditions, we take advantage of sparse classification.
Each of our characteristic projection functions, as described above, is capable of storing any pattern of 255 bits. However, for the task of calibration we restrict this to a continuous block of 1's any where within the domain of the function. We label the lowest positive bit for a particular projection (y, u or v) and class (*COLOR*) as $Min_{proj,COLOR}$. Similarly we label the highest positive bit $Max_{proj,COLOR}$. This means that each characteristic projection function is essentially testing the clause

$$(\, Min_{proj,COLOR} \leq x \,) \wedge (\, x \geq Max_{proj,COLOR}). \tag{6}$$

Therefore each characteristic function may be written

$$\begin{aligned} \texttt{class_}COLOR\,(y,u,v) = {} & (\, Min_{Y,COLOR} \leq x \,) \wedge (\, x \geq Max_{Y,COLOR}) \\ & \wedge (\, Min_{U,COLOR} \leq x \,) \wedge (\, x \geq Max_{U,COLOR}) \\ & \wedge (\, Min_{V,COLOR} \leq x \,) \wedge (\, x \geq Max_{V,COLOR}). \end{aligned} \tag{7}$$

It is therefore this representation, in the form of a decision list (Witten & Frank, 2000) that we expose to the user. A typical calibration file has the following format:

```
Colour_ID_1 Min_Y Max_Y Min_U Max_U Min_V Max_V
Colour_ID_2 Min_Y Max_Y Min_U Max_U Min_V Max_V ...
```

Of course, we are limited in the number of characteristic functions we can apply by the size of the selected data type, as explained above. In our case we are limited to 32 characteristic functions which, of course, may be increased if a larger data type is used. The results of applying one such characteristic function can be seen in the screen-shot in Fig. 4. Here the Y, U and V channels are calibrated separately to produce the overall characteristic function.

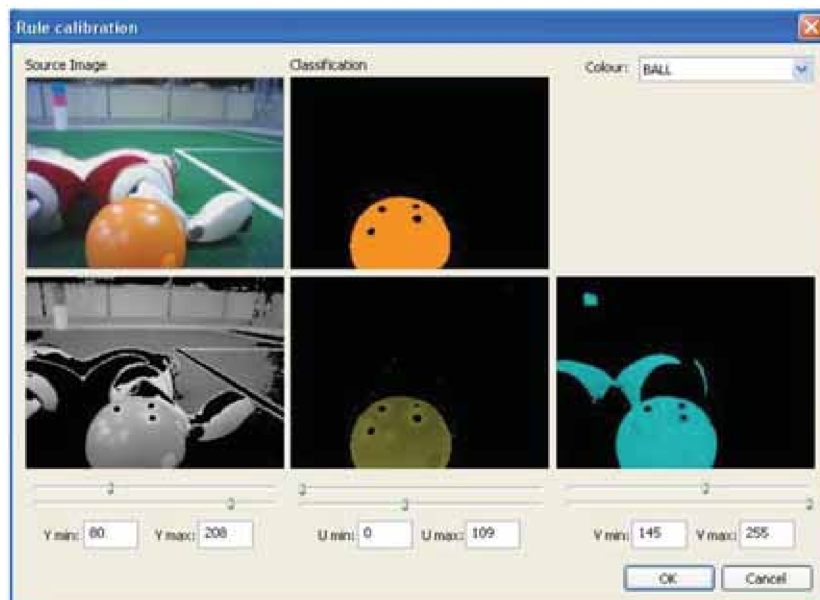

Fig. 4. We may calibrate for each colour by separately examining the Z, U and V components and finding the projections for each characteristic function. This image shows our manual calibration tool allowing the user to select the *Y* (grey-scale in the image), *U* (yellow-scale in the image) and *V* (blue-scale in the image) projections for the class orange separately

Although the rule format restricts us to linear discrimination within the color space, significant flexibility is achieved by the decision list format as Fig. 5 illustrates. Image (a) represents the class orange that we want our classifier to learn. In Image (b) the bounding rectilinear area minimally surrounds the class orange. This is the optimal linear discriminator that completely contains the class, but it does not give a very good solution. There is a large area that our classifier will label orange that is not orange. By using a decision list format we can do much better. We first find a characteristic function for the two shaded areas in Image (c), and label these pixels 'not orange' (i.e. unknown). The while loop in Algorithm 1 will only continue evaluation until a characteristic function returns true. Therefore, if the shaded areas in (c) are tested before the rule in (b), then pixels within them will not be classified as orange even though the linear discriminator in (b) would have classified them so. In this way it is possible to obtain relatively good classifications of colors. Of course, if we are training a sparse classifier instead of a robust one, we will not be interested in a box that completely surrounds the color class. Instead we will want a discriminator that contains the core of each class. In this case linear discriminators are more than adequate. Of course we may require more than one characteristic function to adequately describe the core of each color class (Image (d) in Fig. 5).

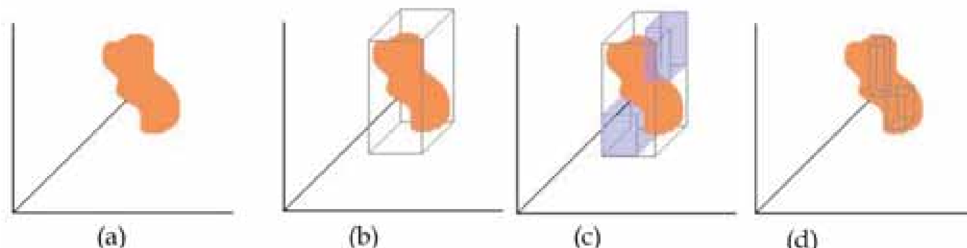

Fig. 5. (a) Colour classes cannot be described by orthonormal rectangular regions in the color space. Therefore (b) classification by linear discrimination is typically bad because it labels many pixels that are not in the class. (c) We may use linear discrimination in combiantion with decision lists to do a better job. If the shaded regions are labelled not in the class, and are higher in the list, pixels within them will not be labelled as belonging to the class. (d) More than one charcteristic function can be used for each colour class. This is particularly useful for sparse classification where only the core of each color is required

4. Optimized Edge Detection

Edge detection is often ignored in dynamic computer vision applications due to the high runtime cost associated with sliding a processing window over an entire image. We have obtained (Lovell, 2007) very optimized methods for edge detection based on effective methods such as Canny's and Sobel's. While our optimizations considerably improve the performance of the Sobel's algorithm, and the resulting algorithm is an order of magnitude faster than Canny's, but it is still computationally expensive. Here we focus on a second alternative that enables us to delay edge detection until it is required (late edge detection). By delaying the edge detection phase we have found it possible to use edge detection as a fundamental tool in our image-processing pipeline because only the areas of the image where edge detection is required will be examined for edges.

4.1 Late Edge Detection

The challenge is to identify interesting sections of the image where edge detection would be useful, rather than apply the difference and window testing of Canny's or Sobel's methods to every pixel of the image. Edge detection is usually an early step in the image-processing pipeline so it is unlikely that we will have a large amount of contextual data on which to base such a decision. But, assuming that we can identify some points inside interesting objects, it is then possible to use edge detection to locate the edges of that object and use them for feature extraction. For now, we will simply assume that we have identified $p = (x, y)$ as a pixel that is contained within an object for which we need to find the edges. We discuss here two techniques depending on the amount of edge information that is required.

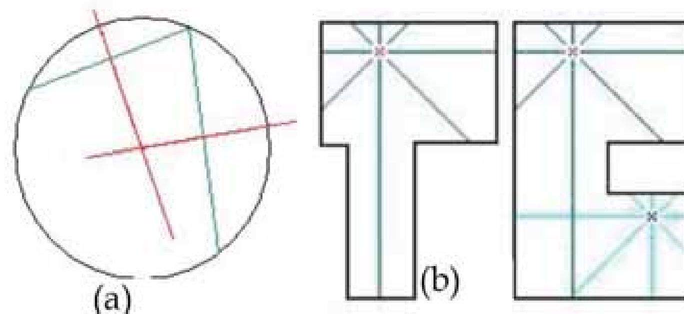

Fig. 6. (a) Vectorization of a circle does not require all the edge points to be known. Any three points on the boundary of the circle can be used to loate its centre by the perpendicular bisectors method as shown in this figure. (b) Patial edge detection on a convex shape (such as the left one in (b)) may not produce enough points to correctly identify and parameterize the shape. If the shape is not convex, further sample points may be needed, or complete edge detection may be requried

Full or complete late edge detection is a technique we use when a complete description of the edge of the object is required. This renders traditional edge detection on a relevant object within the image. However, it is not always necessary to find the complete edge of each object. For example, we are only required to know three points on the edge to find the correct parameterization of a circle (see Fig. 6). In this instance, we use a partial late edge detection that can find n points on the boundary without actually tracing the entire boundary. The difference between the two algorithms is illustrated in Fig. 7. We describe the partial edge detection first, because the complete one will build on some of these techniques.

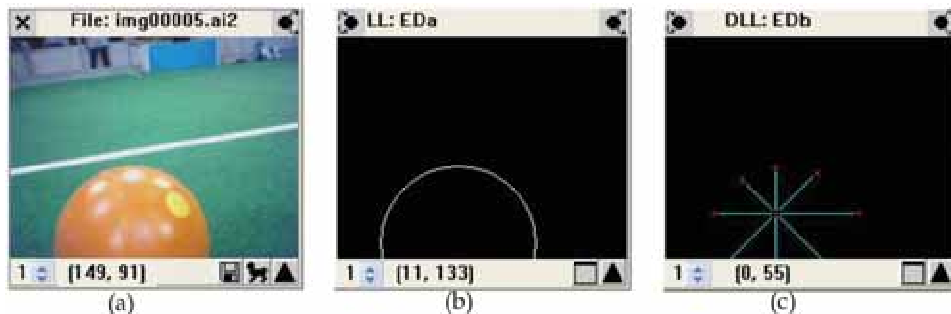

Fig. 7. Full Late Edge Detection on an image (a) results in (b) a full list of pixels that compose the edges in the image. (c) Partial Late Edge Detection locates only a subset of boundary points for an object

4.2 Partial Late Edge Detection

The idea of partial edge detection is that, given some seed point $p = (x, y)$ that we know to be within the boundary of an object, we wish to find a set E of n pixels that lie on the boundary.

To do this we cast n evenly spaced rays out from p, examining each pixel on the ray as we come across it. Each pixel is compared with its neighbors to check the gradient change in intensity. By examining the pixels along a ray in this way, we are essentially performing a Sobel's gradient comparison. Edges are therefore detected well when they are approximately orthogonal to the ray — which is most of the time if the shape is convex.
Of course, if the shape is not convex then we may not be able to gather enough information to parameterize it in this way (see Fig. 6 (b)). In this case, casting rays from a second point p_2 (or more) may sometimes be sufficient for parameterization. More complex shapes will require our second method. By using the Manhattan distance on the color space, we have been able to successfully handle edge detection even in blurry images (Lovell, 2007).

4.3 Complete Late Edge Detection

Sometimes it is not adequate to know only a sample of the boundary points. For example, for vectorisation we require an entire list of spatially connected edge pixels that represent the boundary of an object. Although our method for this full late edge detection is slower than a partial edge detection, it is still significantly faster than traditional edge detection that must be performed on the whole image.

Let $B(p, I) \rightarrow p'$ be a standard border following algorithm that, given a pixel p on the border of an object in a raster image with borders marked I, returns the next pixel around the border of an object p'. Our late edge detection algorithm is then defined by Algorithm 2.

Algorithm 2 Full late edge detection.
Require: A source pixel p that is within the spatial boundary of an object to identify in image (with no marked edges) I.
Ensure: The complete list of pixels E that make up the boundary of the object.
1: Trace any ray from p to find an edge as described in Section 4.2 and label this pixel s.
2: Let the current pixel be c.
3: `while` $c \neq s$ do
4: Apply any edge detection window to locate borders around c.
5: $c = B(c, I)$
6: `end while`

Of course in Line 4 we may use Sobel's window (or optimizations of it (Lovell, 2007)). We show some images in Fig. 8 that illustrate the results of this algorithm. The edge detection is complete in that a full list of pixels that compose the border of a particular object are discovered, but the algorithm does not need to examine any unnecessary pixels to do this. Irrelevant sections of the image are never examined because the algorithm uses a border follower.

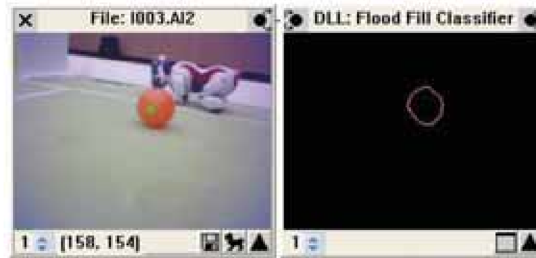

Fig. 8. A full Late edge Detection on a single object within an image reveals the boundary of that object without examining any unrelated areas of the image. In this figure we show a full edge detection on the ball, starting from the pixel indicatedy the green in the source image

One of the problems associated with this technique arises if the edge detector is unable to form closed contours. This issue can be somewhat avoided by a sensitive calibration (that is, one that forms thick edges). However, occasionally we will be forced to abandon an attempt at identifying the edge of the object. By bounding the pixels in the edge of each object we can abort unsuccessful attempts.

4.4 Border Following.

There are many standard border following algorithms that we can apply as *B* in Algorithm 2, all of which are extremely fast ($\Theta(n)$) on the number *n* of pixels in the border). We describe here one of the simplest for completeness. This is a standard algorithm that operates in 8-connected space, but it can be easily modified to work in 4-connected space. Let $D(p, d)$ be a function that returns the next pixel from *p* in direction *d*. Let the directions be defined as in Fig. 9.

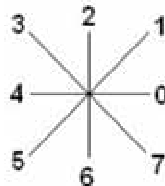

Fig. 9. Direction definitions for the border following method in Algorithm 3

5. Illumination Independence

We now introduce an efficient object recognition system that is robust to changes in color intensity and temperature. Our algorithm will use our edge-detection methods to obtain a description of the boundaries of objects. A list of features in the boundary of the object, plus its color, are usually sufficient for object recognition of landmarks and the ball in RoboCup and for many other object recognition tasks (Lovell, 2007).

Our algorithm will provide this list of features as a basis for object recognition in a wide

variety of illumination conditions without re-calibration. It can do this because it does not rely solely on color classification in order to form blobs. We present two variations of our method. One variation runs extremely quickly but is only able to find simple shapes. The other variation is slightly more processor-intensive but will recognize an arbitrary shape.

Algorithm 3 A simple 8-connected border following algorithm.

Require: Initial pixel p that lies on the border of an object in image (with edges marked) I. The direction dir returned from the previous call.

Ensure: The next pixel p' in a counter-clockwise direction around the border. The direction dir' to use in the next call.

1: Initialise $dir = 7$ on first call.
2: $p' = p$
3: **if** $dir\%2 == 0$ **then**
4: $dir = (dir + 7)\%8$
5: **else**
6: $dir = (dir + 6)\%8$
7: **end if**
8: **while** p' is not on border **do**
9: $p' = D(p, dir)$
10: $dir = (dir + 1)\%8$
11: **end while**
12: $dir' = (dir - 1)\%8$

It is well accepted that edge detection algorithms are far more robust to changes in the temperature and intensity of light than color based segmentation algorithms. This is easy to illustrate. Consider Fig. 10 which illustrates the effect of varying the illumination intensity (a) and temperature (b) of a common scene in RoboCup. Although the edge information in the images is not lost until the illumination levels become extremely low, the robust color calibration becomes useless quite quickly.

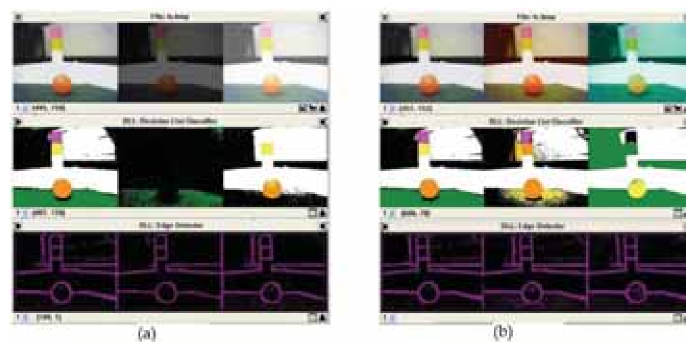

Fig 10. This figure shows the result of varying (a) illumination intensity and (b) color temperature on a common scene in RoboCup. The top rwo of images is the source image, the middle row is a robust classification and the bottom row is our edge detection routine. Note that although variation in illumination conditions is detrimental to color segmentation, it does not particularly affect edge detection

It is apparent then that edge detection, rather than color segmentation, is a better basis for object recognition systems if robustness to dynamic lighting conditions is a requirement. However, edge detection on its own will not yield sufficient information quickly enough. For example, to find the ball in the binary edge images in Fig. 10 we would need to run a circle detection algorithm such as the Hough transform. This would be far too slow for our purposes. Instead, our method works by combining edge detection with the sparse classification technique introduced earlier.

5.1 Building a Sparse Classifier

While it is extremely unlikely that a pixel-color classifier can be built for variable illumination conditions, we may, however, train a sparse one as long as there is not too much variation. Fig. 11 shows how this is possible. As the lighting conditions vary, the perceived color changes in a non-predictable way ((a) and (b)). However, as long as there is some overlap we may classify only the pixels in the overlap section as orange (c). We therefore have found a core class orange that contains pixels that are perceived as orange across both images. The classification itself (c) would be poor if we were to use it for object recognition purposes, but we do not wish to use it directly in this way.

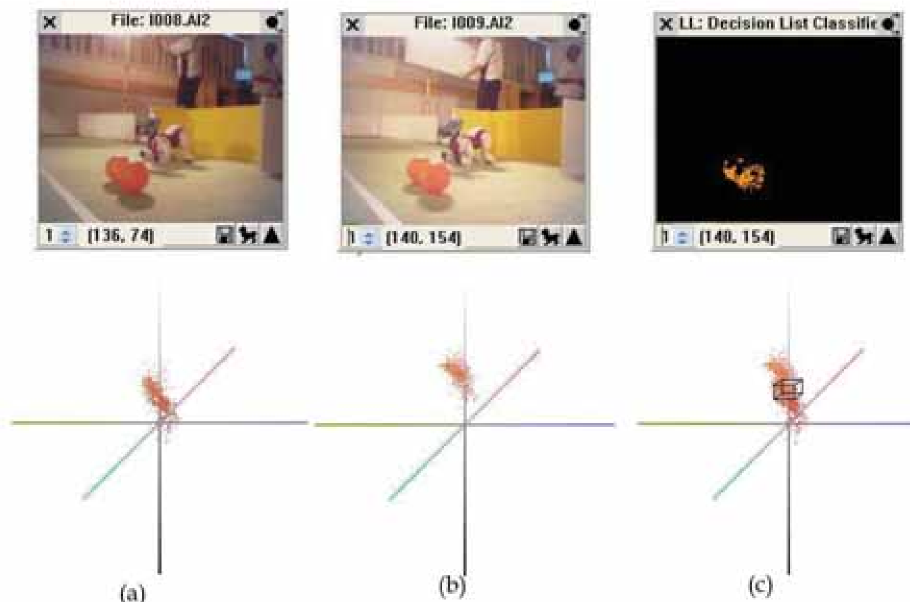

Fig 11. This figure shows the result of varying illumination conditions on the color space. For both images (a) and (b) a perfect calibration was made and every pixel that was labelled orange is plotted in the three dimensional YUV space. Notice that the location of the orange colour class shifts in the space as the illumination changes. We can find the core of the colour class orange (c) by only classifying pixels that are labelled orange in both images. This leads to a poor robust classification, but a good sparse classification

We see now how it is possible to train a sparse classifier for dynamic illumination conditions. We simply widen our set of training images to include many images from different lighting environments and, in the manner described above, train a sparse classifier to label only the pixels that are at the core of each color class. Of course if the lighting conditions are too variable then the intersection operation of the sparse classifier Equation (5) will yield an empty set. In this case we should be less ambitious with our illumination conditions.

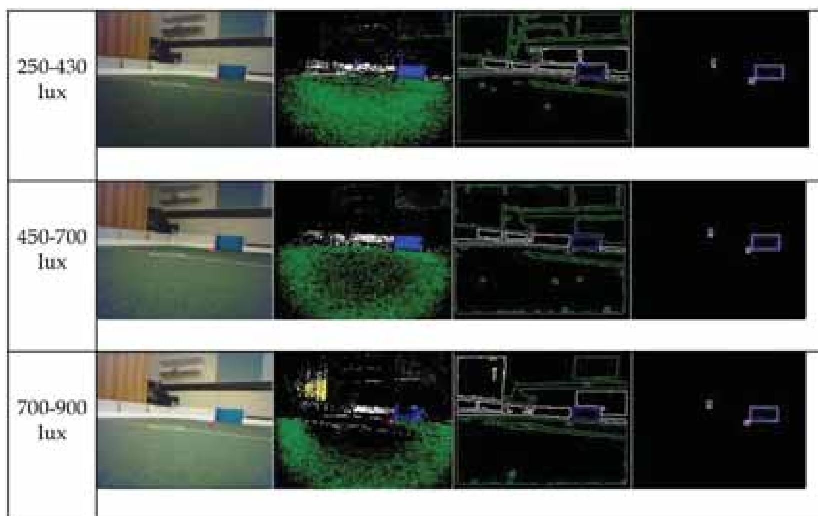

Fig. 12. The basic algorithm for illumination-independent object recognition. A sparse classification (colimn 2) is used along with a border-following algorithm (column 3) to locate regions within the image. These regions can be analyzed for objects of interest (column 4). Note that the system uses the same classification in each illumination condition with accurate results

5.1 Combining Edge Detection with Sparse Classification

One initial algorithm for illumination-independent object recognition is shown in Algorithm 4. In this algorithm we use the list of labeled pixels as seed points to find the border points of each object in the image. Although we can not be sure that every pixel that is part of, for example, the ball, will be labeled as orange, we can be certain that some of them will. Therefore to find the border points of an object we start at a seed point and iterate over pixels in any direction until a previously identified edge is found (Line 4). Once we find an edge we may border trace using the algorithm in the previous section (Subsection 4.4) to obtain the entire list of pixels in the border.

Algorithm 4 Basic illumination-independent object recognition.
Require: An image I.
Ensure: A list of the border points E_o for each object o in I.
1: `EdgeDetect`(I)
2: `SparseClassify`(I)
3: **for** all classified pixels p in I **do**
4: Iterate across pixels in any direction until an edge pixel e is found
5: **if** e does not belong to the border of a known object **then**
6: `BorderTrace`(e) to obtain a list of border points E
7: Assign new object o with edge E, E_o
8: **end if**
9: **end for**

This process renders an image segmentation that works in a similar fashion to other region-growing techniques (Wan, 2003) except that it has the advantage of being illumination-independent. Refer to Fig. 12 where each row in the table shows the process working under a different illumination condition (but with the same calibration file). The first picture is the source image for the row, the second shows the results of a sparse classification step, the third shows the result of Algorithm 4 and the final image shows the object recognition step. In each case the ball, goal and beacon are found correctly, despite the large variation in illumination.

	Component	Average time on AIBO
Our basic pipeline	Sparse classification	1.71 ms
	Edge detection	20.90 ms
	Border following	17.71 ms
	Object recognition	5.87 ms
	Time taken for 30 frames (1s of data)	1385.70 ms
The Bruce et al pipeline	Robust classification	2.42 ms
	Blob forming	9.13 ms
	Object recognition	6.21 ms
	Total per frame	17.76 ms
	Time taken for 30 frames (1s of data)	532.80 ms

Table 2.The runtime cost of basic illumination-independent object recognition is quite high compared to the Bruce et al. pipeline. A large component of this cost is the edge detection and border following steps; therefore, optimizing these steps will improve the runtime cost of our algorithm

The disadvantage of this method is the runtime cost. Refer to Table 2 for a breakdown in the runtime cost of this algorithm, compared to the Bruce et al. pipeline. Most of the execution time is taken in the edge detection step. This edge detection step uses a significantly optimized version of Sobel's edge detection, but remains the main runtime cost of Algorithm 4 (Lovell, 2007). However, in what follows we suggest specialized edge detection

algorithms, rather than a general box to include in the pipeline. The algorithms we will describe significantly improve the efficiency of Algorithm 4 to make it feasible for use in real-time robotic environments.

Notice that Algorithm 4 is not fast enough to analyze 30 fps on an AIBO. In fact, thirty frames analyzed at this speed take 1404.3ms, or just under one and a half seconds. Clearly, a large part of the cost of the algorithm is associated with the edge detection and border following steps. Therefore, we propose two modifications to Algorithm 4 in order to make it execute quickly enough to use in a mobile, autonomous robotic environment.

Both of our alternatives use our late edge detection technique (Section 4). By delaying the edge detection step until it is required, we can perform edge detection only on the sections of the image in which we need it.

6. Detecting Simple Object Boundaries

The first of our alternatives uses the partial late edge detection that we described in Section 4. There are several cases where we do not require a complete description of the boundary of an object in order to parameterize it, as occurs with most simple shapes. The ball in RoboCup is a good example because it is circular. We require only three points on the boundary of the ball in order to apply the geometrical technique of perpendicular bisectors to find the center and radius (Fig. 6). Therefore, we need only to cast three rays in different directions from a seed pixel *p* that we are sure is part of the ball. Of course, we may wish to cast more rays, or to start with more than one seed point, in order to check if we have actually found the ball or just some other object that also looks orange. This is an extremely fast ball-finding algorithm.

Algorithm 5 Efficient rectangle parameterization.

Require: A set P of points that lie roughly on the boundary of a rectangle of unknown aspect and size but known orientation, θ.

Ensure: The parameters of the rectangle R as four corner points $-$ p_{tl}, p_{tr}, p_{bl} and p_{br}.

1: **for** all $p \in P$ **do**
2: Rotate P by angle $-\theta$
3: **end for**
4: Find the bounding, rectilinear rectangle of P and assign to R.
5: Let $maxS_1 = 0$, $maxS_2 = 0$
6: **for** all x in R **do**
7: Let S = the sum of points that lie on or near x
8: **if** $S > maxS_1$ **then**
9: $maxS_2 = maxS_1$
10: $maxS_1 = S$
11: **end if**
12: **end for**
13: Let x_1 = MIN($maxS_1$, $maxS_2$), x_2 = MAX($maxS_1$, $maxS_2$)
14: Let $maxS_1 = 0$, $maxS_2 = 0$
15: **for** all y in R **do**
16: Let S = the sum of points that lie on or near y
17: **if** $S > maxS_1$ **then**
18: $maxS_2 = maxS_1$
19: $maxS_1 = S$
20: **end if**
21: **end for**
22: Let y_1 = MIN($maxS_1$, $maxS_2$), y_2 = MAX($maxS_1$, $maxS_2$)
23: R = { $p_{tl}=(x_1, y_1)$, $p_{tr}=(x_2, y_1)$, $p_{bl}=(x_1, y_2)$, $p_{br}=(x_2, y_2)$}
24: Rotate R by angle θ around the origin

There are other shapes as well that can be found in this way. We present in Algorithm 5 our fast algorithm for determining the parameters of a rectangle provided the angle of orientation is known a priori. If the angle of the horizon in the image is known, then this algorithm can be applied to find (for example) the beacons or goals in RoboCup or a great many rectangular shaped objects in the real world.
We use the method outlined above to find a set P of points that are on the boundary of the rectangle. The points are then rotated in space so that the rectangle is aligned with the axes (Line 2). Each vertical column and horizontal row is then examined to find the lines on which the most points in P lie (Lines 6-21). These lines are labeled as the sides of the rectangle (Line 23). Finally the four corners are rotated back to the frame of reference of the image (Line 24). If the aspect ratio of the rectangle is also known a priori, the algorithm adjusts the rectangle based on the side that had the weakest support value (S). Similar algorithms may be employed to detect any regular polygon.
If enough original sample points are chosen in the initial set P of points, then the algorithm is quite tolerant to noise in the image and even eliminates points where the edge has been determined incorrectly (see Fig. 13 in the next section).

	Component	Average Time on AIBO (ms)
The Bruce et al pipeline	Robust classification	2.42
	Blob forming	9.13
	Object recognition	6.21
	Total per frame	17.76
	Time taken for 30 frames (1s of data)	532.80
Our pipeline using only simple object recognition	Sparse classification	1.71
	Edge point detection	6.71
	Object recognition	1.64
	Total per frame	15.35
	Time taken for 30 frames (1s of data)	460.50

Table 3. Byusing our partial Late Edge Detection in conjunction with the sparse classification, our object recognition pipeline is not only illumination-independent, it executes faster than the Bruce et al. pipeline

6.1 Runtime Performance of Simple Object Detection

In Table 3 we examine the runtime performance of simple object recognition. Note that the two object recognition components in Table 3 should not be directly compared. Much of the work that happens in object recognition in the Bruce et al. pipeline is performed in the shape recognition phase in our pipeline. The most expensive components of this pipeline are edge-point detection algorithms and shape recognition, but the pipeline is still a dramatic improvement on the basic pipeline in Algorithm 4. Indeed, this pipeline is not only

illumination-independant, it is faster than the Bruce et al. pipeline. We could potentially analyze 60 fps at this speed.

7. Detecting Complex Object Boundaries

The second of our alternatives uses the full late edge detection that we described in Section 4.3. The basis of this method was that a point on the boundary of an object would be found in the same way as above, casting a ray from a seed pixel *p* until we reach the edge of the object. We noted in Section 4.3, particularly in Fig. 6, that it might not be enough to simply cast rays from seed pixels to determine the parameters of a complex shape.
We therefore proposed Algorithm 2, a combination of a boundary-following algorithm and a partial late edge detection that could be used to discover and trace the boundary of an object in linear time.

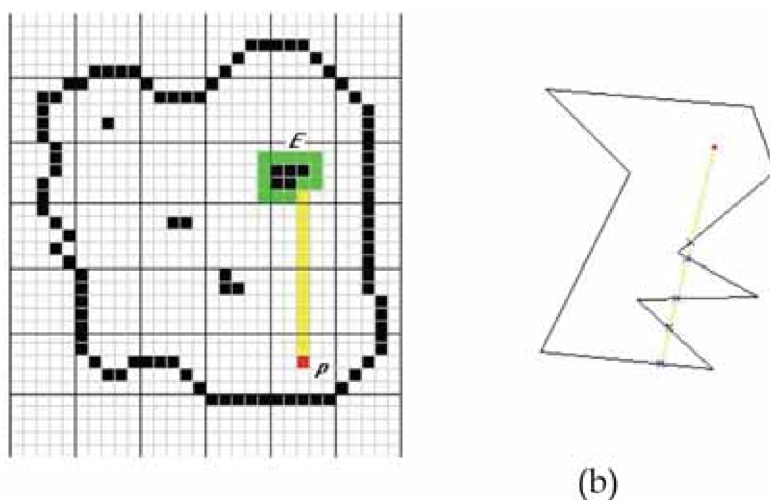

Fig. 13. (a) One of the main roblems with our method is that islands can be found and traced instead of the border of the object. (b) We use a standard polygonal interior test to detect this situation

Algorithm 2 will, under most conditions, return a list *E* of pixels that represents the boundary of the object in which the pixel *p* resides. Due to noise in the image, on occasion, it is possible for *E* to locate an island, rather than the object. Refer to Fig. 13 (a) where the yellow pixels illustrate the ray cast from *p* (the red pixel) and the green pixels indicate the edge, *E*, that has been located. The correct boundary of the object has not been found in this case due to noise in the image.
We may use the standard polygonal inclusion test (O'Rourke, 1998) to detect this situation (Fig 13 (b)).
A ray cast from any point to a point on the edge of a polygon cuts the polygon an odd number of times if that point is inside the boundary of the polygon. We wish to see if our original point *p* is inside the polygon represented by *E*. Therefore the algorithm is simple. As

E is constructed (Algorithm 2), we count the number of times a pixel $q = (x, y)$ is placed in E with $q_x = p_x$ and $q_y < p_y$. If this number is odd then the original point p is inside the polygon E, otherwise it is outside.

Another problem that may be encountered due to noise in the image is that the edge detector does not return a closed contour: that is, there may be gaps in the list of pixels that comprise the edge of the object. If the edge detector is calibrated to be sensitive, (that is, for thick edges), then this is not a frequent problem (Lovell, 2007). When the anomaly occurs it is possible to detect it by bounding the number of pixels that may comprise the edge of an object. In this case, we discard the edge entirely and the object is missed in that frame.

Once the boundary has been correctly determined, it is usually useful to vectorise it. This can be done in linear time (Lovell, 2007). Once a vectorization has been obtained there are several useful and fast analysis algorithms (Lovell, 2007).

7.1 Runtime Performance of Complex Object Detection

Most object recognition tasks will blend a mixture of objects that can be detected using simple object recognition, and objects that must use complex object recognition. Table 4 shows the runtime cost of such a system. Images in these tests were from the RoboCup domain and had a mixture of balls, beacons and goals (representing simple objects) and opponent AIBOs (representing complex objects).

	Component	Average Time on AIBO (ms)
The Bruce et al pipeline	Robust classification	2.73
	Blob forming	8.99
	Object recognition	14.71
	Total per frame	26.43
	Time taken for 30 frames (1s of data)	792.90
	Sparse classification	2.51
Our pipeline	Edge point detection	7.98
	Simple shape recognition	5.06
	Complex shape recognition	10.44
	Object recognition	1.34
	Total per frame	27.33
	Time taken for 30 frames (1s of data)	819.30

Table 4. The final runtime cost of the two pipelines including analysis of both simple and complex objects. Our pipeline executes less than 1ms/frame slower than the Bruce et al. pipeline but provides an illumination independent object recognition system

We make several observations on Table 4, particularly with respect to the introduction of

complex objects (AIBOs). In order to analyze images containing other AIBOs (Lovell, 2007), we must identify their skeleton so the processing that was done on blobs (in the Bruce et al. pipeline) and on complex shapes (in our pipeline) is enough to identify a complete and ordered list of pixels along the border of the AIBOs' uniforms. We do not, however, include a full AIBO recognition algorithm in either pipeline.

This processing represents one extra step for the Bruce et al. object recognition component than for our pipeline. This is reflected in the slower processing time for object recognition for the Bruce et al. pipeline than was seen in Table 2 and Table 3. Our object recognition has no extra overhead because this work is done in the complex shape recognition component.

Different sample images were used for this set of tests (so that AIBOs could be included) so minor variations from the data in the above tables are to be expected. Classification is marginally slower in these images because there are more potential colors to classify and therefore a longer decision list.

We see from this table that our pipeline is essentially equivalent in speed to the Bruce et al. pipeline, being less than 1% slower. Therefore, we have managed to provide an illumination-independent object recognition system for minimal extra cost.

	Object	Occurrences in 300 Images	Recognized (simple)	Accuracy (%)	Recognized (complex)	Accuracy (%)
Stationary Camera (300 Images)	Blue Goal	123	117	95.12%	109	88.62%
	Yellow Goal	119	114	95.80%	107	89.92%
	P/B Beacon	57	56	98.25%	51	89.47%
	B/P Beacon	83	79	95.18%	76	91.57%
	P/Y Beacon	34	31	91.18%	29	85.29%
	Y/P Beacon	62	57	98.39%	54	87.10%
	Orange Ball	212	203	95.75%	192	90.57%
	Red AIBO	87	-	-	63	72.41%
	Blue AIBO	112	-	-	86	76.79%
Moving Camera (200 Images)	Blue Goal	97	86	88.66%	73	75.26%
	Yellow Goal	86	76	88.37%	68	79.07%
	P/B Beacon	52	45	86.54%	31	59.62%
	B/P Beacon	35	29	82.86%	23	65.71%
	P/Y Beacon	39	34	87.18%	27	69.23%
	Y/P Beacon	41	34	82.93%	29	70.73%
	Orange Ball	156	139	89.10%	111	71.15%
	Red AIBO	49	-	-	31	61.22%
	Blue AIBO	54	-	-	29	53.70%

Table 5. The accuracy of our object-recognition system in a set of 500 images: 300 taken from a stationary camera, and 200 contain some degree of blur. The images are taken over a variety of lighting conditions

8. Accuracy of our Method

We have collected 500 images that contain scenes that may be expected in a typical RoboCup game. The image database is divided into two sections. The first 200 of the images are taken from a moving AIBO (and thus contain varying degrees of blur), the other 300 are taken from an AIBO standing still. The images vary in lighting condition (200-1500 LUX, with dynamic shadows) though they are similar enough that a suitable sparse classifier has been found (see Section 5.1).
Table 5 shows the accuracy of our system. There are several things to note here. Firstly, the system performs well — especially given the variable lighting conditions. Even in images taken from a moving camera the system performs well enough to use as the primary sensory input of a soccer-playing robot.
The second thing to notice is that we may determine the extent of the closed contour problem (Section 7) by comparing the accuracy of our system on simple objects with the accuracy achieved when we detect these same objects using our algorithm for complex objects. Typically, we only fail to detect objects due to this problem in less than 10% of images taken from a stationary camera. This accuracy is sufficient for complex shape recognition like AIBO posture recognition [Lovell, 2007]. We fail to detect objects approximately 25% of the time when the images contain blur. This is to be expected — complete edge detection in blurry images remains a difficult problem. Table 5 shows only the positive accuracy (that is, objects in each image that were correctly identified). There were an insignificant number of false positives — less than 10 for all objects in all images — however, this result is not significant to the discussion in this chapter. It is also possible to rule out false positives (Lovell, 2007).

9. Conclusion

We have discussed an approach to object recognition inspired on the vision analysis pipeline. However, the linear organization of the pipeline is a model that propagates the decision of each filter, and therefore, it propagates mistakes. It is natural that after one pass over the pipeline, feedback from the result to one or several of the filters would improve the overall result. For example, finding a large circular orange ball may facilitate finding yellow goals and red AIBOs in the image since now we have information of where the ball is. Also, we have illustrated that not every pixel in the image most be processed by the entire pipeline and thus we can avoid processing some regions of the image.
These remarks suggest two avenues for expanding our work. First, we can have some areas of the image significantly advanced on stages of a vision pipeline whose results may be input to other areas or early filters. Second, running the pipeline with parameters that emphasize speed but coarse results may enable further later executions of the pipeline on the same image with feedback information and adapted parameters. Having a very fast and robust pipeline here means that as CPU-speeds increase, we can run them very effectively. Notice that as resolution of the frames increases linearly, the number of pixels increases quadratically, similarly, as the frame rate increases linearly, the number of pixels that needs analysis increases quadratically. Executing a fast pipeline like ours will enable more reliable and robust systems under even larger illumination variations as we move into faster processors and more reliable hardware.

10. References

J. Bruce, T. Balch, and M. Veloso. (2000). Fast and inexpensive color segmentation for interactive robots. In *Proceedings of the International Conference on Intelligent Robots and Systems*, (2061–2066). IEEE Computer Society Press. ISBN: 0-7803-6348-5.

S. Chalup, R. Middleton, R. King, L. Li, T. Moore, C. Murch, and M. Quinlan. (2004) The NUbots' team description for 2004. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII*, Lisbon, Portugal, pages CD–Rom Proceedings. Springer-Verlag. ISBN: 3-5402-5046-8.

J. Chen, E. Chung, R. Edwards, N. Wong, E. Mak, R. Sheh, M. Kim, A. Tang, N. Sutanto, B. Hengst, C. Sammut, and W. Uther. (2003). A description of the rUNSWift 2003 legged robot soccer team. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII*, Padua, Italy, pages CD–Rom Proceedings. Springer-Verlag. ISBN: 3-5402-2443-2.

K. Gunnarsson, F. Wiesel, and R. Rojas. (2005) The color and the shape: Automatic on-line color calibration for autonomous robots. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX*, Osaka, Japan. Springer- Verlag. ISBN 3-540-35437-9

A. Halme, K. Koskinen, V-P. Aarnio, S. Salmi, I. Leppnen, and S. Ylnen. (2000). Workpartner — future interactive service robot. In *Proceedings of the Millennium of Artificial Intelligence Conference, 9th Finnish Conference on Artificial Intelligence*, pages CD–Rom Proceedings. Finnish Artificial Intelligence Society. ISBN: 9-5122-5128-0.

A. Kak and N. DeSouza. Robotic vision: What happened to the visions of yesterday? In *Proceedings of the 16th International Conference on Pattern Recognition*, (839–847). IEEE Computer Society Press, 2002. ISBN: 0-7695-1695-X.

N. Lovell. (2007) *Machine Vision as the Primary Source of Input for Mobile, Autonomous Robots.* PhD thesis, Griffith University, Nathan, 4111, QLD, Australia, 2007. Available www.cit.gu.edu.au/˜s2130677/PhDthesis/nLovell.pdf.

Y. Ogihara, Y. Shibata, H. Najima, K. Kii, K. Oda, and T. Ohashi. (2005) Asura: The kyushu united team 2005 in the four-legged robot league. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX*, Osaka, Japan, pages CD–Rom Proceedings. Springer-Verlag.

J. O'Rourke. (1998). *Computational Geometry in C*. Cambridge University Press, U.K.

T. Röfer, R. Brunn, S. Czarnetzki, M. Dassler, M. Hebbel, M. Jungel, T. Kerkhof, W. Nistico, T. Oberlies, C. Rohde, M. Spranger, and C. Zarges. (2005) GermanTeam 2005. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX*, Osaka, Japan. Springer-Verlag. ISBN 3-540-35437-9

J. Shaik and K. Iftekharuddin. (2003). Automated tracking and classification of infrared images. In *Proceedings of the 2003 International Joint Conference on Neural Networks*, (1201–1206). IEEE Computer Society Press. ISBN: 0-7803-7899-7.

M. Shaw and D. Garlan. (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, U.S.A. ISBN: 0-1318-2957-2.

S. Shimizu, T. Nagahashi, and H. Fujiyoshi. (2005) Robust and accurate detection of object orientation and id without color segmentation. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX*, Osaka, Japan. Springer- Verlag. ISBN 3-540-35437-9

M. Veloso, S. Chernova, C. McMillen, P. Rybski, J. Fasola, F. vonHundelshausen, A. Trevor, S. Hauert, and R. Espinoza. (2005). Cmdash05: Team description paper. In

Proceedings of RoboCup 2005 – Robot Soccer World Cup IX, Osaka, Japan, pages CD–Rom Proceedings. Springer-Verlag. ISBN 3-540-35437-9

S. Wan. Symmetric region growing. (2003) *IEEE Transactions on Image Processing*, 12(9):1007 1015. ISSN: 1057-7149.

I. Witten and E. Frank. (200) *Data Mining – Practical Machine Learning Tools and Technologies with Java Implementations.* Morgan Kaufmann, U.S.A. ISBN: 1-5586-0552-5.