Vision Software for a Humanoid Soccer Robot

David Prasser

19th October 2001

43 Aberfoyle St,
Kenmore,
Brisbane QLD 4069

19th October, 2001.

Professor Simon Kaplan,
Head of School,
School of Information Technology
and Electrical Engineering,
University of Queensland,
St Lucia QLD 4072.

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Electrical Engineering, I present the following thesis entitled “Vision Software for a Humanoid Soccer Robot”. This thesis project was conducted under the supervision of Dr Gordon Wyeth.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

David Prasser.

Acknowledgements

Gordon Wyeth for starting the whole GuRoo project.

Mark Chang for his endless advice and of course the SH4 board.

The other vision people: Andrew Blower, Shane Hosking, and Adrian Ratnapala for their ideas and advice.

Mark Venz and Damien Kee for providing the RoboLab Public Library.

Damien Kee, Ilan Lacish and Mark Venz again for their help with LYX and LaTeX.

The rest of the GuRoo team: Andrew Smith, Emanuel Zelniker, Andrew Blower, Damien Kee, Mark Wagstaff, Anthony Hunter, Shane Hosking, Nathaniel Brewer, Timothy Cartwright, Jarad Stirzaker, and Bartek Babel.

And everyone else in the robotics lab for making it such a relaxed and friendly lab.

Abstract

This thesis describes the initial work in developing a software system to provide visual information for a humanoid soccer playing robot. The robot is intended to operate in a colour coded environment. The system is to report the location of significant objects for playing soccer - the ball, goals, edgelines and opponents. In addition to this it is important for the robot to determine its own location and orientation. The software runs on a Hitachi SH4 processor board and is intended to use an OV7620 CMOS camera to provide YUV or YC_BC_R images.

Several approaches were investigated before the final system was designed. The system has a colour detection first stage that uses a UV lookup table. This is followed by a run length based grouping algorithm that constructs objects from the colour detected image. Finally simple heuristics are used to reject poorly defined objects.

The code was trialled on an SH4 processor with a resolution of 240 by 180 pixels and operates at frame rate of 10 frames per second performing colour segmentation and object classification. The image size is limited by the amount of memory on the processor board. Further increases in speed would be possible by transferring parts of the code from the SH4 processor to a field programmable gate array that is also on the board.

The use of colour lookup tables combined with a row based object growing gives a fast method for robotic vision that will be suitable for humanoid robot soccer.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Robotic Vision

One of the significant problems in robotics is extracting information from the environment. All but the most primitive industrial robots have some form of external sensing, such as sonar, infrared, or contact switches. The most versatile method is through computer vision, an algorithmic analysis of the digital output of a camera.

The problem then becomes one of constructing a model of the real world within the computer from the digital image received from the camera. A large amount of work has been done in this field of image processing. Approaches and algorithms for image processing can be roughly divided into two sections: real time and off-line image processing. Typically real time image analysis is done more crudely than offline analysis because of constraints on the amount of computational time that can be allocated for each image. Most robotics applications require real time processing, although applications that are not time critical may have a low frame rate.

1.2 RoboCup

The Robot World Cup or RoboCup is an international competition in which various types of robots compete against each other in games of soccer. The goal of RoboCup is to promote research in robotics while at the same time providing a series of complex problems for robotics and artificial intelligence development. The new focus of RoboCup is humanoid robots with several humanoids being demonstrated at the 2000 RoboCup in Melbourne and competitions with humanoids planned for 2002[20]. The University of Queensland is building a robot, the GuRoo, to compete in future humanoid competitions [21]. Among the many systems that need developing for the GuRoo is a vision system to allow the robot to locate the ball, opponents, and other important data. The GuRoo robot will provide a platform for testing embedded vision systems.

1.3 GuRoo, the University of Queensland Humanoid Project

To represent the University of Queensland in the humanoid league at future RoboCups a robot, the GuRoo[1], is being designed. The GuRoo is intended to be approximately 1.2m high and be capable of autonomous movement, with on-board power and computer systems. Additionally the Guroo will have a fully functional upper body, including arms and a head that can tilt and pan (Fig 1.1).

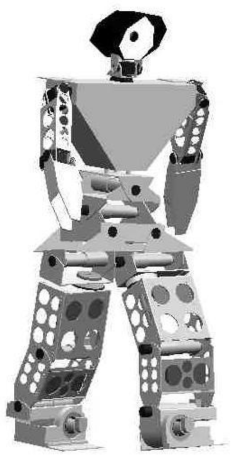


Figure 1.1: The Guroo
The GuRoo is the a humanoid robot currently under development at the University of Queensland. [21].

The GuRoo's central computer will be a Compaq iPaq handheld computer connected to the drive control processors via a Controller Area Network (CAN) [12]. The iPaq will be responsible for the generation of walking patterns and gameplay intelligence. Also on

[1]GuRoo, the **G**rossly **U**nderfunded Roo. The suffix Roo is traditionally used for the University of Queensland's robot soccer teams.

board the GuRoo will be a Hitachi SH4 processor that analyses and interprets the output of a CMOS camera mounted in the robots head[5]. The vision processor passes on its output into the expansion slot of the iPaq. This SH4 processor will be used for testing the vision software developed in this thesis.

1.4 Robocup Vision

The RoboCup scenario will be used to provide some examples of the tasks an embedded vision system would need to be able to complete. The vision system is responsible for determining the location of the robot on the playing field as well as locating the ball, the goals and any opponents. It must be able to do this fast enough and accurately enough to give the robots the ability to determine the velocity and to intercept the ball. Latency is also an issue as the longer it takes to process a frame the less valid it becomes.

To ensure that the real time vision is possible the RoboCup competition enforces certain requirements that make colour segmentation possible. For example the edge lines of the field are white and the ball is orange. This makes colour detection a useful way of segmenting the objects of interest in the field.

There are two approaches to the way vision is used robotic soccer at the moment: global vision and local vision. Global vision uses a camera external to the robots which contains the entire playing field within its field of view, this technique is used only be the small size league robots. The alternative is a local vision system where the camera is mounted on the robot. This technique immediately leads to several complications:

- The camera is not in a fixed position or orientation, this means that the robot must be able to perform transformations to convert from image coordinates to world coordinates.
- The camera will not have a complete view of the area the robot is operating in. Typically RoboCup robots that use local vision have an inter-robot communication strategy to compensate for this problem.
- The shapes of objects change as they rotate making feature based object recognition more difficult. The ball being rotationally invariant is immune from this problem.
- The size of objects change which also makes object recognition more difficult.

The robots requirements for a high frame rate can also lead to significant trade offs in terms of both reliability and accuracy with local vision. In the context of humanoid robot soccer local vision is required.

1.5 Outline of Thesis

Introduction: An introduction to the robotic vision and its application to RoboCup. This chapter also discusses the the GuRoo and its planned vision system.

Literature Review: A review of current practice in image processing and robotic vision techniques, with emphasis on RoboCup robot.

Problem Specification: This chapter gives a detailed description of the requirements and goals of the GuRoo's vision system. This section will also describe the available resources for the software such as camera and processor types.

Initial Approaches: A number of early solutions to the problems outlined in chapter four are described, with particular emphasis on why they were never developed further.

Final SH4 Software: This chapter describes the process used to transform the camera output into a coded representation of each colour. The methods used to extract the location and type of objects from the thresholded image will be described.

Results: In this chapter the success of the software will be evaluated in terms of frame rate, accuracy, and robustness.

Conclusion: Final results of the project are summarised and future improvements are described.

Chapter 2

Literature Review

2.1 Robotic Vision

The initial step in most robotic vision applications is segmentation, in which the original image is segmented into different regions containing objects of interest[11]. After segmentation more complex algorithms can be used to determine the relevance and type of each object as well as its centroid and any other parameters of interest.

The very first stage of segmentation is usually to simplify the image by a process known as thresholding. Thresholding reduces the amount of information in an image by converting each pixel into some sort of symbolic code representing what type of pixel it is[11, 14, 10]. For example in text analysis a greyscale image could be converted into a binary image which is true for pixels that are part of the text and false for pixels that are part of the background.

2.1.1 Greyscale Thresholding

Thresholding is a simple technique for segmenting images. The most primitive example of thresholding is converting a grey scale image into a binary image composed of background and foreground objects. For instance all pixels above a particular level of brightness are foreground while those below are background. The difficulty lies in finding the threshold value to ensure that there are no errors [11]. One technique for selecting the correct thresholds is to use a histogram of the greyscale values [11, 14]. This system can be expanded to segment the image into more than just background and foreground by using ranges instead of a single value.

2.1.2 Colour Thresholding or Colour Detection

Another method of segmenting an image is through the colour information received from a colour camera [2, 3, 14]. In effect this is like thresholding but with three values, for instance red, green, and blue. Histograms can again be used to determine the ranges for thresholding [6]. However the computational complexity has now been increased by a factor of three.

2.2 Morphology Theory

Mathematical morphological is a useful tool for preprocessing images [10]. The morphological operators treat binary images as sets which are members of a two dimensional coordinate space.

The simplest two operations are erosion ($\ominus$) and dilation($\oplus$). These operations have two operands, an input (A) and a structuring element (B). Erosion results in groups of pixels becoming smaller while dilation causes groups of pixels to become larger[10].

Erosion is defined mathematically as equation 2.1, where $(B)_x$ is the element B translated by x. Effectively an erosion results in a set of points for which B fits entirely inside A. An example of bitmap undergoing erosion is shown in figure 2.1. Erosion can be used to remove small groups of noise pixels from a colour detected image[10].

$$A \ominus B = \{x \mid (B)_x \subseteq A\} \tag{2.1}$$

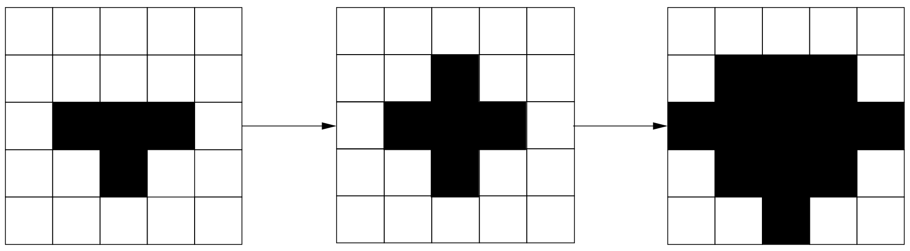
Dilation causes a cluster of pixels to grow in size according to the structuring element B2.1. The set theory definition is shown in equation 2.2 where $\widehat{B}$ is the reflection of B around its origin. The result of a dilation is the set of points that B would occupy as it is moved around the image but still overlaps by at least one pixel with A. Dilations can be used to link together elements that have been incorrectly separated by the colour detection or thresholding processes[10].

$$A \oplus B = \left\{x \mid \left(\widehat{B}\right)_x \cap A \neq \phi\right\} \tag{2.2}$$

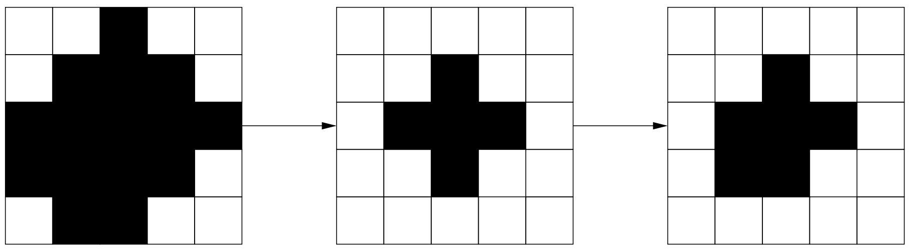
The final two common morphological operations are opening and closing. An opening ($\circ$) is simply an erosion followed by a dilation with the same structuring element (equation 2.3). The erosion removes noise pixels and the dilation restores the the thinning caused by the erosion. A closing ($\bullet$) is effectively the opposite of an opening being a dilation followed by an erosion (equation 2.4). The effect of a closing is to link together pixels that are separated by a small distance by dilation. The erosion then restores the objects to their original size, keeping the new linked regions intact[10].

$$A \circ B = (A \ominus B) \oplus B \tag{2.3}$$

$$A \bullet B = (A \oplus B) \ominus B \tag{2.4}$$



(a) Dilation



(b) Erosion

Figure 2.1: Morphological operations.
The input is on the left and the output is on the right. The structuring element is in the middle.

2.2.1 Edge Detection

One technique commonly used for determining the shape of objects is edge detection. A sudden change in the image gradient, ie when one pixel is significantly darker than some of its neighbours, corresponds to the edge between two objects or one object and the background [19]. Typically an approximation to the gradient operator such as Sobel edge detectors are used[11, 19]. The Sobel operators shown in figure 2.2 provide an indication of edge strength when convolved with a greyscale image[10, 14]. Finally the gradient is thresholded to eliminate small variations that are caused by noise[19]. The result is a bit plane containing ones that correspond to sharp changes in brightness in an image.

−1	0	1
−2	0	2
−1	0	1

(a) x gradient

−1	−2	−1
0	0	0
1	2	1

(b) y gradient

Figure 2.2: Sobel Edge Detectors
The Sobel edge detectors calculate the change in brightness when they are convolved with a greyscale image. The X gradient operator (G_X) calculates the change in brightness in the X axis and Y gradient operator (G_Y) calculates the change in the Y direction.The sum of the two operators will provide the edge strength at the point the operators are applied [10, 14].

2.2.2 Hough or Radon Transform

The Hough or Radon transform is a powerful method for converting an image of edge pixels into a set of lines. A line can be represented by the equation $\rho = x\cos\theta + y\sin\theta$ [14]. A two dimensional array of accumulator cells is created with one dimension being the line parameter ρ and the other θ. For each pixel in the edge map a set of parameters can be developed that represent all possible lines passing through that pixel. The corresponding accumulator cells for these parameters are increased by one. After each pixel has been processed in this way the cells with large values correspond to dominant lines in the edge map image[14].

An alternative approach with exactly the same result is the Radon Transform. The radon transform uses a rotating vector in the centre of the image (figure 2.3). At each angle of rotation the edge detected pixels are projected onto the vector. If the intensity of the projection is plotted against the distance along the vector and the rotation of the vector then the same output as the Hough transform is obtained.

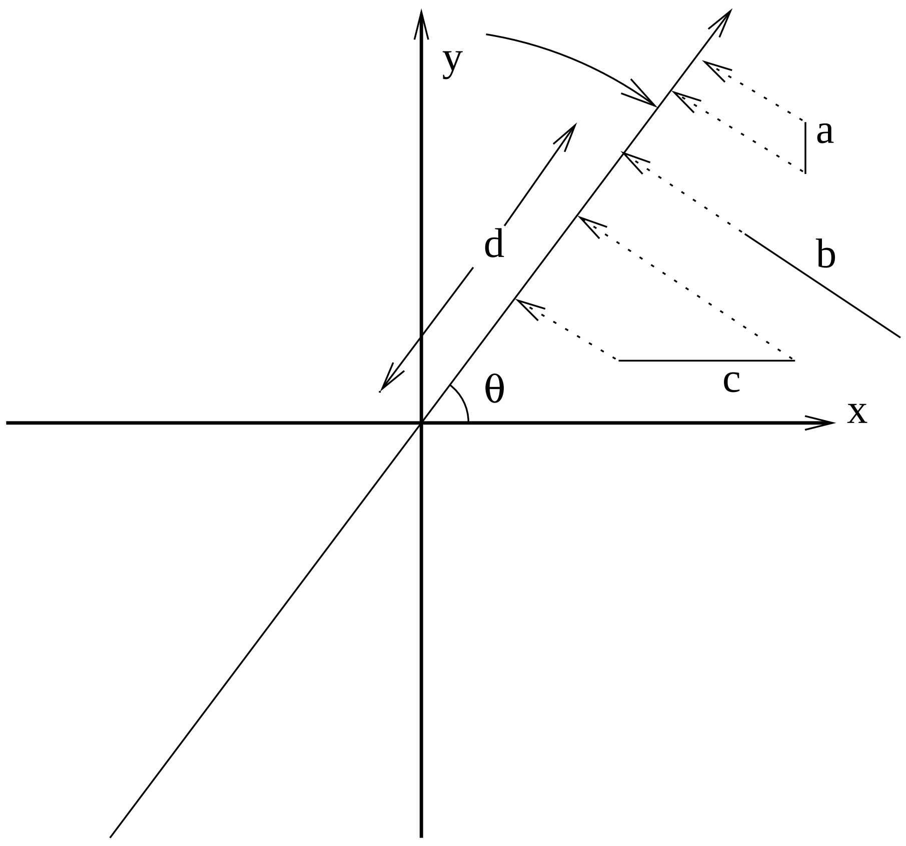


Figure 2.3: Radon Transform

As the vector rotates around the origin the summated projections of each edge pixel onto the vector are recorded. When the vector is perpendicular to the direction of a group of edge lines the output is a maximum for that group of edges. For example the group of pixels labelled b in the above diagram would be giving its maximal progression at this rotation. The maximum would be at distance d along the vector.

2.3 Robotic Vision in Robocup

Real time image processing for robotics is a particularly difficult area of work. The vision problems expected in Humanoid local vision are very similar to the problems encountered in the local vision version of the small and medium size leagues. The fundamental system used for segmentation in these leagues is colour[2, 3, 4].

Typically in RoboCup competitions the colours of all significant objects in the competition are standardised and known before hand, for example the ball is orange and the goals are blue and yellow. In global vision systems the robots usually make use of coloured markings to distinguish between robots. In localised vision however the robots are restricted to being black in colour[16, 17]. The Medium Size league uses colour tags on the robots to provide a mechanism for robots to distinguish between each other[16].

2.3.1 YUV Colour Space

Many systems use the YUV colour space to perform colour segmentation[2]. The YUV is a different interpretation of colour to the more familiar RGB colour space, where a pixel is composed of varying strengths of red, green and blue light. In the YUV system Y represents the intensity of the light, while U and V are the blue and red chrominance respectively [14] A representation of YUV data is shown in figure A.1. The conversion from RGB to YUV is ideally performed in the camera.

The advantage of the YUV colour space is that changes in brightness theoretically only affect the Y value. This means that colour detection can be performed using only the U and V values. The U and V coordinates are almost independent of lighting changes.

2.3.2 HSI or HSV Colour Space

Some robots use a different colour space known as HSV or HSI (hue, saturation, brightness). This colour space offers better separation between different colours but at the cost of slower processing as the camera output must be transformed into the HSV colour space[3, 4]. In HSV colour format the hue parameter describes the colour of the pixel while the saturation value represents how white this hue is. Clearly the brightness describes how bright the pixel is. The advantage of HSI is that is fairly close model of a humans perception of light[14, 10]. A simple way to speed up the transformation is to only calculate the pixel's hue. The hue will describe the colour of the object and may be enough to distinguish between all colours except black and white (which can be easily done in RGB or YUV space) [4].

2.3.3 After Thresholding

After thresholding the locations and type of all significant objects must be extracted from the picture. These objects include the ball, opponents, walls, goals and in some cases special markers for localisation (used in the Sony Legged league). Approaches to this are varied but are usually simple and optimised for speed.

2.3.3.1 Localisation of Objects

A simplification that can be made to the process of transforming two dimensional data to three dimensional world coordinates is to realise that in robot soccer the ball and other objects can be almost guaranteed to be on the ground. This makes the transform solvable with monocular vision without having to use information relating to the size of the object [2]. The AgiloRoboCuppers team use this to help discard redundant information, by calculating the size of each object and comparing it to the expected size based on its distance calculated by trigonometry if there are significant discrepancies then the object is ignored[2]. The trigonometry is encoded as a lookup table to reduce the processor load.

2.3.3.2 Localisation of the Robots

In the Sony Legged league there are colour marker posts that allow the robots to determine their position by triangulation [20]. In the other leagues however there are no such conveniences, and odometry information is not sufficient to retain a correct idea of the robots place in the world. The usual approach is to use the edge lines of the field to update the robots estimate of its location. The edge lines can be acquired by looking at transitions from white to green codes in the colour detected image[2] or by using a linear edge operator[3]. Linear regression or a Hough transform can then be used to find the lines inclination, from this the robots position can be determined[4, 2, 3].

2.4 CM Vision

A good example of a fast colour based system for image analysis is the CM Vision software library developed at Carnegie Mellon University[8, 7]. It internally uses either a YUV or HSI colour space and transforms RGB inputs to YUV. Colour thresholding is performed using a colour cube in YUV space. A colour cube is box-like region in the three dimensional colour space. Pixels within this region can be tagged as being one type of coloured object while pixels outside this region are not.

Rather than use a series of comparison operations to determine whether or not a pixel is within the colour cube CM Vision uses a lookup table. Instead of a three dimensional $256 \times 256 \times 256$ table CM Vision uses three vectors one for each axis. Each colour coordinate is looked up in its respective vector and the result is ANDed together. This technique reduces a 16 MB lookup table to a 768 byte set of three tables. The cost of this system though is that it requires three memory lookups instead of one and can only represent rectangular regions in the colour space[8, 7].

After colour detection the system converts the data to a run length encoded data set. In run length encoding horizontal runs of pixels of the same colour are recorded simply as the number and colour of the pixels. From this the software performs four connected region growing. Four connected or four neighbour refers to the four pixels directly above, below and to the sides of the pixel in question. Initially each run length element is tagged as a the start of a blob and the software runs through each row comparing it with adjacent rows. Any run that is adjacent under four connectivity to the current one has its pointer updated. Two runs through the image are required to group all the runs into distinct regions. The final stage involves grouping similar regions based on colour, proximity and density, that links any regions that have been accidently separated[8, 7].

2.5 ViperRoos

The University of Queenslands local vision team, The Viperroos faces similar problems to the GuRoo's vision system. The system used last year was based on YUV transformed data from an RGB camera[9].

The Viperroos use a two dimensional lookup table in UV space to distinguish between all the colours with the exception of black and white. The Y value is thresholded into three sections: black, colour, and white. Pixels that fall into the colour region are then thresholded via UV lookup.

The Viperroos' software then forms regions from the colour detected image and calculates the regions' centres. To avoid the multiplications and divisions involved in calculating a mathematical centroid the Viperroos define the centre to be the geometric centre of the blob.

The Viperroos currently use an SH3 processor running at 104 MIPS which produces around 12 frames per second. This year the camera will be upgraded to one with YUV output and the processor will be replaced by an SH4 with 360 MIPS. These changes should move the Viperroos into the 25-35 frame rate range[9].

This information is of particular interest as the GuRoo will be using the Viperroos new processor board and cameras.

Chapter 3

Problem Definition

The vision software should be useful to both the ViperRoos and the GuRoo as both face similar problems at the early stages of the software, ie fast colour segmentation. Additionally they will be using the same hardware which makes using the same software much more attractive.

The GuRoo's only external sensor is the camera mounted in its head. From this sensor all of the information needed must be derived. At present requirements for accuracy are unknown as the robot has not been constructed or the environment in which it it is to operate specified. The medium and small size leagues are used as starting points for developing specifications for the vision system.

3.1 Object Recognition and Self Localisation

The location of the ball, obstacles, edge lines and goals needs to be determined relative to the robot. From this information the robot can work out its own location in the environment as well. No specifications for accuracy exist because the nature of the environment in which the robot operates is undefined.

Using the rules for other local vision leagues for RoboCup leads to the conclusion that the environment will be extensively colour tagged with significant objects having distinct and defined colours (Table 3.1)[17, 16]. Obviously the software must be capable of colour recognition to meet these requirements.

A capability original suggested for the humanoid vision system was to be able to provide information suitable for the control systems of the robot to use as part of the stability system. This would require calculating velocities and positions of the head with respect to the environment.

Object	Colour
Field/Operating Surface	Green
Ball	Red
Edge of Field Lines	White
Opponents	Black
Goals	Blue or Yellow

Table 3.1: Colour Coded Environment

In the colour coded environment the robot operates in each object of interest has an assigned colour.

3.2 Frame Rate and Resolution

The University of Queensland's other local vision robots, the ViperRoos face similar problems and uses the same hardware as the GuRoo. The ViperRoos' code is currently running on an SH3 processor with a RGB camera and has a frame rate of about 16 frames per second (fps) or 62.5 ms with a resolution of 32 by 128 pixels[9]. About 20 ms of time per frame is used to perform a costly RGB to YUV colour space transform. In terms of pixels per second the SH3 operates at around 64 kilopixels per second. The costly YUV conversion can be eliminated by using a a YUV output camera reducing the processing time for a frame by 20ms. This would increase the number of pixels analysed by the system to 96.4 kilopixels per second. The ViperRoos' new SH4 processor will run at about three times as fast as the old processor giving a speed of approximately 300 kilopixels per second.

Any software system should at least run through the pixels at the same rate as the old ViperRoos software if not faster. For general three dimensional vision instead of using the ViperRoos letter box type image (32×128) an image with the usual $1\frac{1}{3} : 1$ aspect ratio of a computer screen should be used instead (640×480).

3.3 Hardware

The hardware consists of three physically separate elements the camera board, processor board and the iPaq (Fig 3.1). Communication between the boards is run through a field programmable gate array (FPGA). Data flow is one way although there is the possibility of using the iPaq to alter the program settings on the SH4[12].

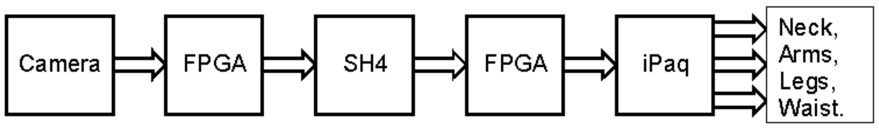


Figure 3.1: Hardware Block Diagram of the Humanoid.
This block diagram shows the flow of information through the humanoid's computer system. Data flow to and from the SH4 is through the FPGA allowing reconfigurable connections between input/output pins [5, 12].

3.3.1 Camera

The camera that will be providing images for processing is an Omnivision OV7620 CMOS camera[5]. There are several reasons for using this particular camera. Firstly it is a CMOS camera which means that it has a digital output and low current consumption both of which are necessary for mobile robots. Secondly the OV7620 has on board subsampling. The previous ViperRoos system performed subsampling inside the SH3 processor, in other words a full resolution picture is transmitted to the processor which discards the majority of the data. Subsampling by the camera reduces the amount of unnecessary data flowing across the bus to the processor. Finally and most importantly it can produce a YUV output[13, 5]. This removes the costly RGB to YUV conversion stage from the software.

3.3.2 Processor Board

The processor board is an based around a Hitachi SH4 microprocessor. The SH4 is capable of running at up to 360 million instructions per second (MIPS) as opposed to the SH3 board it replaces which ran at 104 MIPS[9]. Aside from this approximately threefold increase in speed the SH4 processor board also has 512 KB of static ram (SRAM) as well as static-dynamic RAM (SDRAM). Also this board is equipped with a Spartan field programmable gate array (FPGA) which sits between the SH4 and the camera. The FPGA gives the option of carrying out some processing on the data before it reaches the SH4. It is also planned to use the FPGA to buffer data from the camera as it comes into the SH4 [5].

Chapter 4

Initial Approaches

There were several abortive attempts at solving the problems outlined in chapter 3 before the techniques finally used were developed. All of these techniques where based on YUV images which where trialled in MATLAB.

4.1 Colour Detection

The original segmentation system used separate histograms of the red and blue chrominance. From these histograms the upper *x*% of the red chrominance and the lowest *x*% of the blue chrominance were used as the thresholds for the red ball. The parameter *x* must be adjusted to produce reasonable results. The difficulty with this technique is that the image must be histogrammed and the thresholds recalculated for each image. Secondly while this technique is good at finding red it is not so effective at locating pixels belonging to the ball as it does not combine information from the U and V coordinates. A faster solution is to use lookup tables.

4.1.1 Lookup Tables

A system similar to the CM Vision lookup table was trialled. The CM Vision system uses three vectors to describe rectangular three dimensional regions in YUV space. The problem with this technique is that adding adaptive Y thresholds is difficult and that the rectangular regions in UV space appeared to be less useful than arbitrary regions.

4.2 Orientation

To locate the robot in the in space it was originally planned to make detailed use of the edge lines of the field. By using edge detection techniques an image showing the edge lines can be produced. From this several methods could be used to calculate the line descriptors of the edges. One approach would be to use the Radon or Hough transform (2.2.2), but it was feared this may be too computationally expensive. So a different approach using edge following was trialled.

4.2.1 Edge Detection

The use of traditional edge detection operators such as Sobel or even Canny[14] detectors was avoided for several reasons. Firstly the the operators would require a large number of calculations to performed per pixel (at least four for a 2×2 operator). Secondly the results of these edge detections would be a set of edges for all significant changes in brightness while only the edges of the field lines need to be segmented. Instead of using a mathematical edge operator some other techniques were tried:

- Eroding the segmented white objects in the image and then XORing this with the original segmentation. This produces the set of pixels that are the boundary of all white objects. The problem with implementing this approach is that the entire boundary of the white objects are detected, while only the lower part is needed.
- Creating an edge map of all the white colour coded pixels that are directly above a green coded pixel. This works quite well and would be the best method for determining the edge lines if the field is segmented. In the final approach though the green area of the image is not segmented (Chapter 6).

4.2.2 Edge Following

The edge following system starts from a random point around on the edge map and and begins tracing the edge. As the algorithm moves from pixel to pixel the direction is encoded and stored in a null terminated list (a chain code). As the algorithm moves between pixels in an eight neighbour manner there are eight possible directions to be encoded 4.1. After traversing a set of edge pixels in a straight line the start and end coordinates of the line will have been found.

Direction	Code
$\rightarrow$	1
$\nearrow$	2
$\uparrow$	3
$\nwarrow$	4
$\leftarrow$	5
$\swarrow$	6
$\downarrow$	7
$\searrow$	8

Table 4.1: Eight neighbour direction codes.

The edge following system represents the direction from pixel to pixel using these numerical codes.

As the algorithm moves along the system it uses simple heuristics to determine whether or not the line it is following is a straight line. While tracing a line the algorithm permits only two adjacent directions to be followed. This detects if the line doubles back or significantly changes direction but does not detect a line that only changes direction by 45°or less (figure 4.1). A solution to this problem is to develop an approximation to the gradient and use probability techniques to detect gradient changes[15]. This area of research was abandoned when it was found that the colour detection was not able to provide a straight edge line for a pixel by pixel approach like this.

4.3 Optical Flow

Optical flow is a technique for measuring the movement of pixels from one frame to the next[11]. Optical flow could provide some indication as to the velocity and stability of the head. However it is computationally expensive and was dropped at an early stage when gyroscopes were announced for internal sensors of the GuRoo. Because of the planned internal sensors and concerns with speed and latency within the vision system the use of optical information to assist with the control or balance of the GuRoo was abandoned.

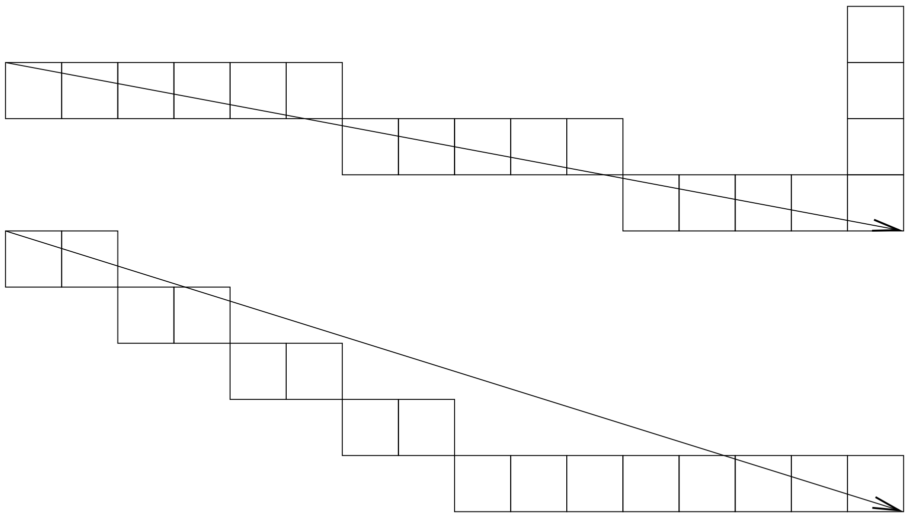


Figure 4.1: Chain Code Line Description

The line following correctly detects the the straight line segment of the upper group of pixels but does not detect to the change in gradient of the lower line. The start and end points of the groups represented by the arrows show what the computer believes is a completely straight line.

Chapter 5

Final SH4 Software

In this chapter the final software running on the SH4 processor is described. It consists of several stages that execute one after another: colour detection; morphological operations; run length encoding; grouping; and object analysis. The software was prototyped in MATLAB and then recoded in C for a PC platform. Once the code was debugged and optimised it was ported to the SH4 using the GNU C compiler for the SH series[1].

5.1 Colour Detection

In the initial colour detection stage the YUV colour image is converted into a new image that contains eight bit codes representing what type of colour class each pixel belongs to. The colour codes used for humanoid vision are listed in table 5.1. The colour codes are set as powers of two so that each class of objects has its own bit plane. In other words the output of the colour detection can be thought of as eight single bit planes with the same width and height as the original image. As a result the system is able to detect eight different colours.

5.1.1 Basic Colour Detection

The final technique is based on a two stage system that first uses the pixel brightness to classify the pixel into one of three classes: black; white; or other. This is done with two simple thresholds, pixels below one threshold are black while pixels above the other threshold are white. The other pixels between the two thresholds are candidates to be some other colour (yellow, green, blue) and require another level of attention to determine which. Colours apart from black and white are determined by a lookup table with U and V coordinates.

The Y thresholds are recalculated for each frame as offsets from the sample mean of the image. The lookup table values are constructed from a trainer program (section 5.6). The result of this process is that each YUV pixel is mapped to a single value describing the colour class of the pixel. The simplified implementation of this algorithm is shown as algorithm 1.

Algorithm 1 Basic Colour Detection Algorithm

```
mean ← mean of y values in image
low threshold ← mean - low offset
high threshold ← mean + high offset
for each pixel
     read Y
     if Y < low threshold then
         output ← BLACK
     else
         if Y > high threshold then
             output ← WHITE
         else
             read U
             read V
             output ← lookup table[U][V]
```

Object	Colour	Code
?	Any other	0
Field	Green	1
Wall/Edgeline	White	2
Ball	Red	4
Opponent/Obstacle	Black	8
Blue Goal	Blue	16
Yellow Goal	Yellow	32

Table 5.1: Colour Codes

After colour thresholding the output image contains these coded values describing each pixels colour. There are two remaining colour codes (64 and 128) that could be used to represent more colours.

5.1.2 Fast Colour Detection

The colour detection process can be accelerated by reducing the number of pixels that the operation is performed on. Instead of processing every pixel only every fourth pixel is used. If there is a change in output between last two pixels tested then the algorithm backtracks and processes the pixels in between. Otherwise it is assumed that these are all of the same value (algorithm 2). This increases the speed of the algorithm but also increases the number of pixels that are misclassified (Table 6.1).

The misclassification is caused by the fact that this algorithm can skip over small groups of pixels or small gaps between pixels (figure 5.1). In fact the algorithm cannot guarantee the segmentation of any object smaller than the subsampling size (4 pixels). However because of the backtracking the actual location and general shape of a segmented object is not damaged.

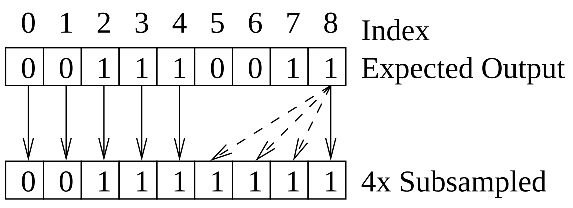

0 1 2 3 4 5 6 7 8 Index
0 0 1 1 1 0 0 1 1 Expected Output
0 0 1 1 1 1 1 1 1 4x Subsampled

Figure 5.1: Subsampling
The first line shows the output of pixel by pixel colour detection. The second line shows the effect of 4× subsampling with backtracking. The 0th and fourth pixels are examined first. The output is different so the 1st, 2nd and 3rd pixels are also tested. The final (8th) pixel is read, because it is the same as the previous subsampled pixel (4th) so the intervening pixels are kept as 1 without ever being thresholded.

5.1.3 32 Bit Operation

As the SH4 is a 32 bit processor reading 24 bit YUV colour information is not particularly efficient because of problems with word alignment and the fact that multiple reads are required. As can be seen in algorithm 1 either one or three memory reads are made per pixel. A much more efficient implementation can be made by reading the pixel as 32 bit value (figure 5.2). Each pixel then needs only one memory read. Furthermore the Y value thresholds can just be scaled and compared to the whole 32 bit pixel as the Y value is in the most significant byte (ignoring the unused byte). Also instead of using U and V as array indices (table[U][V]) which leads to the computer calculating the address as

Algorithm 2 Fast Colour Detection Algorithm

```
end_address ← image_size - CD_SUBSAMPLE
index = 0
while index < end_addr
  next_index ← index + CD_SUBSAMPLE
  output ← colour_detect(next_index)
  if output = last_output
    each output between index and next_index <- output
    index ← next_pixel
  else
    colour_detect each pixel between index and next_index
    index ← next_index
```

$table + U \times 256 + V$ the low 16 bits of the pixel data can be used as a one dimensional index. This is the only stage where the 32 bit architecture required change from the original code.

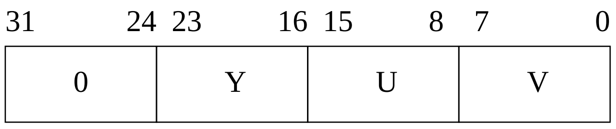


Figure 5.2: 32 Bit Data Storage

The YUV colour information is padded to 32 bits from 24. The magnitude of the 32 bit value approximately 2^{16} times the Y value of pixel. The lower 16 bits are used as the index into the lookup table.

5.2 Morphological Techniques Used in the Code

Morphological operations can be used to decrease the amount of noise in the image after the colour detection phase. To remove the noise quickly only an erosion is used as opposed to an opening. A 1×3 structuring element is used to clean up the colour detected data. The 1×3 element was chosen because it was the minimum size necessary to give good noise rejection.

A simple implementation of 1×3 erosion is shown in algorithm 3. This algorithm makes use of the fact that each colour code is in a separate bit position so an erosion can be thought of as happening on eight bitmaps in parallel. This implies the AND in algorithm 3 is a bitwise AND instead of a logical AND.

Algorithm 3 Simple 1×3 erosion.

```
for i = 1 to size of image -1
  output[i] = input[i-1] AND input[i] AND input[i+1]
```

The morphological system can be sped up by reducing the number of memory accesses. It is clear from algorithm 3 that each memory location is accessed three times, so by keeping previously read data it is possible to reduce the memory reads by three. The C code for this erosion is shown as algorithm 4.

Algorithm 4 Fast 1×3 erosion.

```
i ← 0
for y ← 0 to height
  out[i] ← 0
  i++
  a ← input[i-1]
  b ← input[i]
  c ← input[i+1]
  for x ← 1 to width - 4 step x+=3
    out[i] ←  a AND b AND c
    i++
    a ← input[i+1]
    out[i] ← b AND c AND a
    i++
    b = input[i+1]
    output ← c AND a AND b
    i++
    c ← input[i+1]
    next x
  input[i] ← 0
  i++
  input[i] ← 0
  i++
next y
```

5.3 Grouping

The grouping algorithm converts the colour detected pixels into a list of structures describing each four connected group of pixels. There are two passes in this algorithm, the first pass turns the colour detected image into a list of run length encoded (RLE) elements. These elements are then used to construct a set of blob objects that describe large blobs found in the image. The advantage of using RLE is that the grouping algorithm will be simpler if it operates on blocks of pixels instead of individual pixels. The other advantage is that the RLE compressed data will be convenient for transmitting to the iPaq, so a colour detected image can be displayed on the iPaq's screen[12].

5.3.1 Run Length Encoding

The first stage of the grouping algorithm is to convert from an array of pixels to a null terminated list of run length encoded elements. The run length encoding elements contain the following data members:

begin: The x coordinate corresponding to the beginning of a run.

end: The x coordinate of the end of the run.

y: The y coordinate of the run.

colour: The colour of the pixels in the run.

tag: The number of the run counting from the top left corner.

blobpointer: A pointer to a blob structure (section 5.3.2).

The run length encoding process creates a list of runs in memory that encapsulates each horizontal block of pixels. Unlike normal run length encoding this systems constrains the runs to appear on one line. The tag number is set to be the number of the run counting from the top left hand corner of the image. The tag value zero is reserved for terminating the list of runs. The blobpointer member is set to null when a run is created.

5.3.2 Blobs

A blob is an four connected region of pixels of the same colour. At present a blob contains the following data members:

area: The number of pixels in the blob.

colour: The colour code of the pixels in the blob.

xmax, xmin, ymax, ymin: The bounding dimensions of the blob.

Blobs are the output of the grouping process and the input to the final analysis stage of the vision process. Originally the sum of the moments of each pixel in the blob was stored within the blob structure so that the exact centroid of the blob could be recovered. To reduce the computational load this feature was dropped.

5.3.3 Grouping

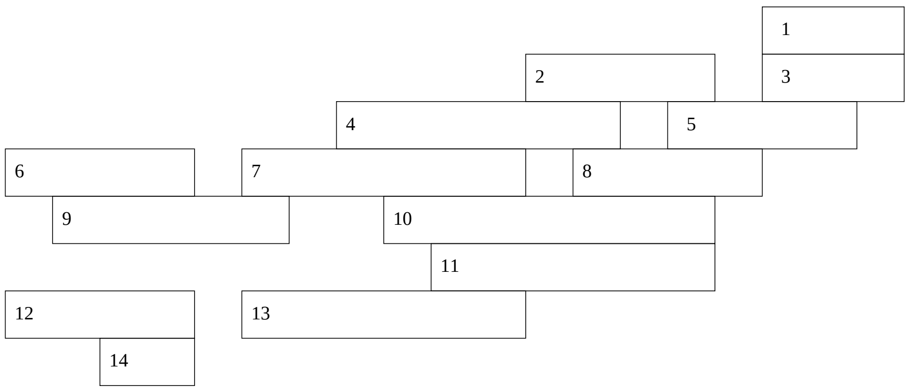
After run length encoding blobs in the image are formed through a grouping process. Each run is examined one at time and compared to the runs on the following row (algorithm 5). Rows that are four connected to the current row and the same colour are part of the same blob object. The basic approach to constructing these blobs is shown in algorithm 6. If both run elements have their blob pointers at null then a new blob object is manufactured for them and their pointers set to it. If only one of the two run elements has a non-null blob pointer than the empty pointer is set to point to the valid blob and the blob parameters are updated. The final case when both of the run elements have set blob pointers then the newest blob is deleted and both of them are set to the older blob.

Algorithm 5 Grouping

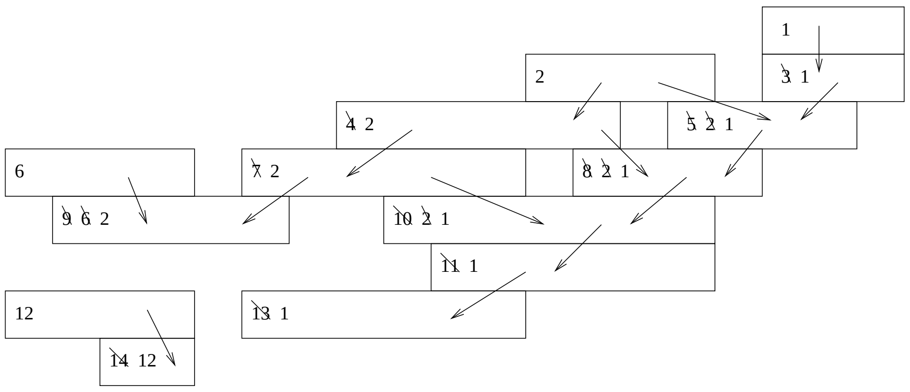
```
r ← start of RLE list
while r.tag ≠ 0
  n ← next RLE element for which n.y ≠ r.y
  if n.y = r.y
    for r ← each RLE element from r to n
      match(r against the RLE elements starting from n)
  r ← r+1
```

Algorithm 6 Matching

```
r is the element being compared to the next row of elements
n is the first element of the next row such that n.y = r.y + 1
rowbelow ← n
while rowbelow.y = r.y + 1 {
  if r and rowbelow overlap in four connected space
    if rowbelow.tag > r.tag {
      if r.blobpointer = NULL
        create a new blob and attach it to r
      merge rowbelow.blobpointer with r.blobpointer
      rowbelow.blobpointer ← r.blobpointer
      rlowbelow.tag ← r.tag
    } else {
      if rowbelow.blobpointer = NULL
        create a new blob and attach it to rowbelow
      merge rowbelow.blobpointer with r.blobpointer
      rowbelow.blobpointer ← r.blobpointer
      rlowbelow.tag ← r.tag
    }
  rowbelow ← rowbelow + 1
}
```



(a) Run length encoded data before grouping.



(b) Propogation of tag numbers through run elements.

Figure 5.3: Run length encoded elements.

Each run element has a tag number which is originally set to the index of the run **(a)**. Figure **(b)** shows the RLE elements after the grouping algorithm has passed through. Each element is matched against the elements in the row directly beneath it. When the elements are four connected the tag number is used to determine which element inherits from the other, the arrow indicating the direction of inheritance. Every RLE element connected by an arrow therefore contributes to the area and bounding rectangle of the blob.

The tag member of the run structure is used to determine which blob is deleted when regions are merged. Smaller tag numbers indicate that the blob associated with it is older. The merging process is simply a matter of increasing the bounding rectangle and area to accommodate the information from the new run or blob.

The grouping algorithm does not produce information showing which runs are members of each blob. The only information that is produced is the bounding rectangle and the number of pixels in the blob. While further passes of the grouping system would eventually be able to provide the exact shape of the blob there does not appear to be any need for this information.

5.4 Analysis

The final stage of feature extraction is the rejection of incorrect matches caused by errors in the first colour segmentation stage. The analysis stage examines the list of blobs and produces a set of objects that the software considers to be the best candidates for each type of object.

The first stage is to determine the approximate height of the wall. The area above the wall can then be ignored. This will remove the possibility of false matches in the area above the wall.

The wall object is the largest white blob in the image. Any object higher than a fixed offset above the wall object is ignored. The blue and yellow goals are likewise defined as the largest blue or white objects except that the must be in the valid region described by the wall. All of these objects must also be over a minimum size or they are not detected at all. This prevents a few noise pixels from being detected as an object.

Obstacles are defined as all black objects in the valid area of the screen that are above a minimum size and a list of obstacles is constructed containing all of these objects.

The ball is a special case in that it is rotationally invariant so it should maintain a fixed unity aspect ratio from any view point. Therefore the difference between the width and the height can be used as a simple test for the ball. This rejects red objects that do not have the correct dimensions for the ball.

5.5 Localisation and Line Detection

The two most powerful cues for localisation are the blue and yellow goals. From the angle of the goal and knowledge of the approximate height of the robot an idea of the robot's position on the field can be calculated. Another source of information for localisation are the edge lines. A fast way of obtaining orientation from the edge lines is to look for horizontal lines. An efficient method to do this is to use a Radon transform that only looks for lines that are horizontal.

Resuming the discussion in 2.2.2 it is clear that to only look for horizontal lines the projections need only be calculated when the vector is vertical. This is equivalent to simply summing the number of edge pixels in each horizontal row. The summation of edge pixels can be stored in a one dimensional array giving a profile of edge strength related to y coordinate. A significant horizontal line in the image will then translate into a large spike in the profile. A maximum detection system can then be used to find this point.

The maximum detection system is quite simple. The maximum point of the profile is first found. If this maximum is greater than a minimum threshold value and also greater by a minimum value than the second largest value in the profile than it is considered to correspond to a horizontal line. Aside from the detection of these cues for orientation no further work has been done in the field of localisation.

5.6 Training

The trainer lets the user provide representative samples of each colour class and construct a UV lookup table from these samples. The user can select a rectangular region of the image and specify it as being of a certain colour.

The selected region is passed to a two dimensional histogramming algorithm that counts the occurrence of each combination of U and V coordinates. This histogram is consequently of size 256×256. This histogrammed data is then thresholded and assigned a code value by the user to produce a lookup table for one colour class. The user can interactively make further histograms of other colour classes and superimpose them onto the lookup table. This results in table like figure A.2**(a)**.

Finally to link up the regions in the table and account for any sort of colour shift the whole lookup table is dilated with 3×3 a cross shaped mask (figure A.2**(b)**). As each bitplane is dilated separately it is possible for the lookup table to contain two colour codes for the same U and V coordinate. As this would result in a pixel being detected as two colours the dilation is post processed to select only one colour code for each cell. Firstly if the cell in the table already contains a colour value then that cell is not permitted to be changed by the dilation. This prevents one colour from overwriting another colour during the dilation. Remaining overlaps are arbitrated by order of priority (table 5.2).

The finished lookup table can be trialled on a test image which duplicates the SH4 vision code to the point where it can show the location of segmented objects.

Priority Number	Colour Code
1	Red
2	Black
3	Blue
4	Yellow
5	White
6	Green

Table 5.2: Dilation Priority

The lookup table dilation uses this priority list to ensure that each position in the lookup table corresponds to only one colour code.

5.7 Memory Organisation

The image processing software requires a large amount of data to be stored within the 512 KB of on board memory. The YUV image; colour detected image; and the eroded image will use a large part of the memory. In addition to this there is also a large amount of symbolic information to be stored such as the run length encoded elements and the blob structures. Memory is allocated to all of these elements using fixed size buffers. Buffers are used instead of of dynamically allocated data structures such as linked lists for several reasons, the chief being that a buffer is simpler to move through. These buffers use up about 400 KB of memory leaving the remaining memory for the program space, local variables and the stack.

Buffer	Size	Memory
YUV Image	240×180	168.75 KB
Lookup Table	256×256	64 KB
Colour Detected Image	240×180	42.2 KB
Eroded Image	240×180	42.2 KB
RLE Elements	3072	72 kB
Blobs	512	12 KB
Objects	64	768 Bytes
Edge Profile	180	720 Bytes
Total		401.8 KB

Figure 5.4: Memory Usage in the SH4.

The remaining 110 KB of the 512 KB of memory is used for the program, local variables and the stack.

Chapter 6

Testing and Results

6.1 Testing Methods

As there was no digital camera input for testing the code with the SH4 pictures were taken with a commercial handheld Casio camera. These 640×480 RGB colour images were converted to YUV format and subsampled to 240×180 using MATLAB and saved as raw binary files suitable for download to the SH4 board.

The SH4 code's performance was tested by using a serial port program to download and upload data from buffers on the SH4. The frame rate can be measured by toggling a LED on the SH4 board every time a frame is processed. The frequency of the LED, which is twice the frame rate, can be measured with an oscilloscope.

6.2 Training and Lookup Table

The training software was used (section 5.6) to construct a lookup table for detecting red, blue and yellow. The white and black regions of the image are detected by thresholding the Y coordinate. These thresholds are set at ± 50 above and below the brightness mean which is recalculated for each new frame of data.

The final lookup table shows the location in UV space of each of the three colours of interest (See figure A.3). The horizontal axis corresponds to increasing red chrominance from left to the right and the blue chrominance increases down the vertical axis. As expected then the region that detects blue is located at the bottom of the image and the red region is located on the top right of the image. It is interesting to note that there is a relationship between the red and blue chrominance of the ball. Roughly speaking as the red chrominance of the ball increases the blue chrominance decreases. This relationship is preserved because the lookup table uses arbitrary shapes as opposed to rectangular regions.

6.3 Colour Detection

6.3.1 Reliability and Robustness of Colour Detection

Two examples of the colour detection system in action can be seen in figure A.4. It is clear that in general the performance of the colour detection stage is fairly good however, there is a problem detecting yellow in the image (figure A.4 **(b)**). This is caused by problems with the brightness thresholding of the image. Basically the Y values of yellow approach those of white leading to confusion as to the difference between white and yellow. Other less bright colours such as red, blue or green are generally detected well.

Aside from misinterpreting yellow as white, the detection of black and white is very successful. This indicates that readjusting the Y thresholds for each frame is an intelligent strategy, especially as the lighting level within the environment can change dramatically.

Another problem encountered during colour detection is that particularly dark shades of green can be detected as red (figure A.6). This leads to extra red in the colour detected image that causes problems later in the image processing system. It appears these problems are caused by a poorly constructed red region in the colour lookup table. Redefining the colour table to be more conservative in red, ie making the red region smaller in the table, may solve this problem.

6.3.2 Speed of Colour Detection

The software was tested on the SH4 processor board and the frame rate recorded for a few test images. The performance of the fast colour detection system at various rates of pixel skipping is given in table 6.1. The fast colour detection by pixel skipping gives reasonable accuracy with a general speed saving of about 12 ms when thresholding every fourth pixel. This is a acceptable error for reducing the speed by approximately 25%.

The speedup is not as large as expected, probably because the branching interferes with the pipeline of the SH4. A subsampling rate of about 20 gives the fastest speed and increasing the subsampling rate beyond 20 does not reduce the processing time. The limiting factor on the subsampling though is that the system is not guaranteed to detect any object narrower than the sampling interval.

Subsamping	Time (ms)	Pixels in Error
none	40	0
6	23.2	331
5	25.2	251
4	28	240
3	32.8	230
2	35.6	117

Table 6.1: Subsampling speed and error
This is the time taken to perform colour detection on a particular 240×180 pixel image. The figure pixels in error is the number of pixels that are different to the colour detected image created with no subsampling. In general terms the error is particularly small (331 errors is 0.8% error for a 240×180 image).

6.4 Morphological Operations

The removal of noise by morphological operations using a 1×3 erosion was generally successful particularly in removing small amounts of red noise. The benefits of the noise reduction can be seen in figure A.5.

The slow erosion process listed in algorithm 3 takes approximately 25 ms to complete. The erosion faster algorithm (4) only reduces the execution time by three milliseconds. A better method for implementing an erosion is outlined in section 7.1.3.

6.5 Run Length Encoding

Run length encoding is another costly operation taking 49 ms to encode the test image into about 180 run length elements. Clearly further operations that use the RLE data will operate quickly as the amount of data to process has been dramatically reduced. Also it is apparent that the buffer allocated for the RLE data is much larger than it needs to be, as space has been allocated for 3072 elements. The reason for this is that if the colour detection fails badly or is miscalibrated the number of RLE elements can become very large and potentially overwrite other sections of the program.

6.6 Grouping

The grouping stage which converts the RLE elements into blobs executes quickly and efficiently as it uses symbolic data as its input rather than an image array. Under the current grouping system an isolated RLE element will not become a blob because the system only creates a blob when it combines two or more RLE elements. This effectively means that there is no need for a morphological erosion in the vertical direction.

One minor bug detected in the code was when a blob was merged with itself the blob was deleted, this was corrected by preventing the blobs from being merged with themselves.

6.7 Blob Analysis and Object Detection

Blob analysis is the most difficult and error prone section of the system. The system can easily make errors, usually these are failures result in no object being detected rather than an imaginary object being found. Two successful examples of the output are shown in figure A.7. The shadow under the ball is detected as an obstacle.

The detection of the goals is particularly successful although that is not a great accomplishment as there should only be one goal in any image and they are large objects. Even though yellow is not particularly well segmented, the large size of the object is sufficient to allow simple detection of the yellow goal. Obstacles are also easily segmented for the same reasons. Although an occasional lighting problem causes a few parts of the field to be detected as black causing imaginary black objects to appear.

The difficulty in the image segmentation process is the red ball. Usually the source of error is poor detection of red, for example figure A.6. The rejection of red noise is not particularly successful at the present stage causing the blob analysis stage to fail to correctly locate the ball. Another source of error is that sometimes the software decides that lines on the field such as the centre line are in fact the edge line. This incorrect decision causes the algorithm to ignore the majority of the image.

6.8 Line Detection

The process for detecting horizontal lines is quite successful at rejecting inappropriate lines (figure 6.1). As the system will not report a line unless the maximum of the edge profile is significantly greater than the second largest value most errors are avoided. For these trials the minimum acceptable peak on the edge profile is 35 and the minimum difference between the maximum and the second largest peak is 17. The software is however unable to differentiate between the lines in the middle of the field and the edge lines of the field. The process of developing the edge profile is slow taking 24 ms to complete, but the peak detection stage was practically instantaneous.

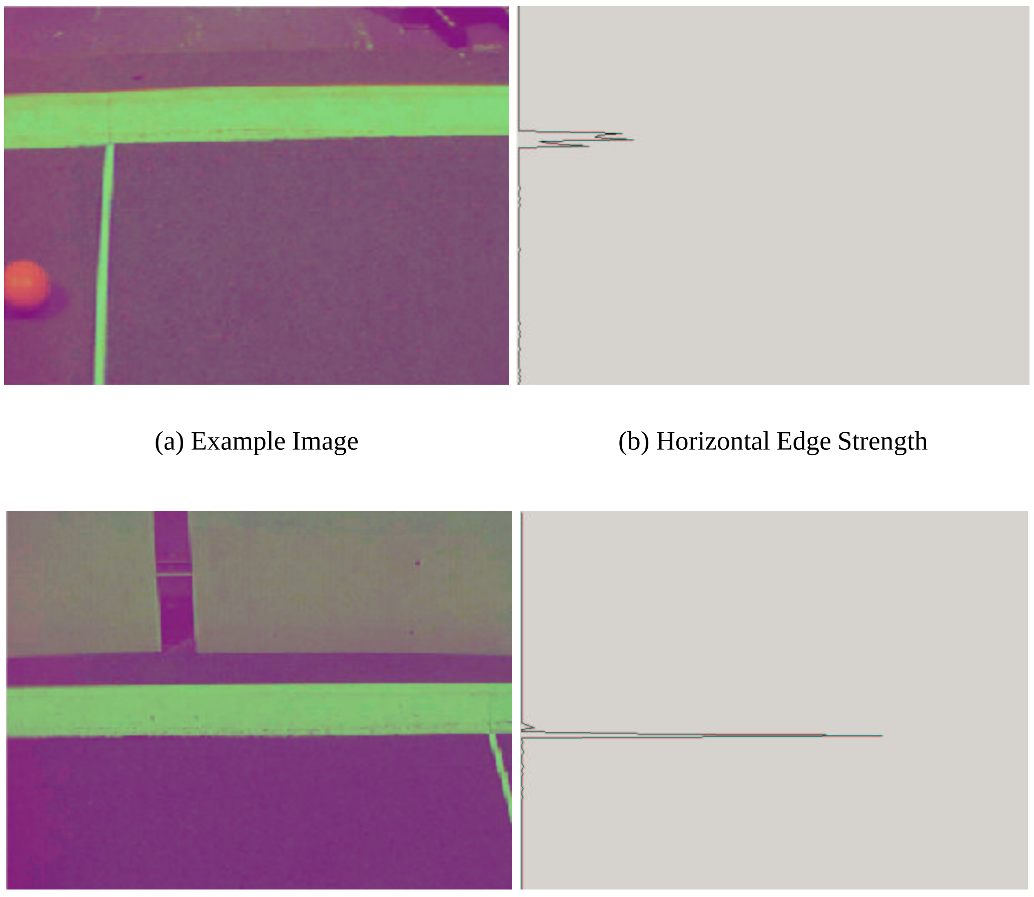
The problem with the line finding at the present stage is that it does not use employ any information from other stages of the process. In particular during the blob analysis stage the bounding rectangle of the edge line is found. If the search was limited to finding horizontal lines in this region the line finding would be faster and less likely to make mistakes.

6.9 Total Processing Time

In total the system operates at 10 fps without the edge detection code in place or at 8 fps with the edge detection code. The breakdown of time spent in the various stages of the image processing system is shown in table 6.2. It is clear that the parts of the system that operate on symbolic data execute much faster than those parts which use image buffers. The code then operates at 348 kilopixels per second which is more than expected (section 3.2).

Task	Time (ms)
Colour Detection	28
Erosion	22
RLE	49
Grouping	Negligible
Analysis	Negligible
Edge Detection and Summation	25
Edge Analysis	Negligible
Total Time	124 ms

Table 6.2: Breakdown of Processor Time



(a) Example Image (b) Horizontal Edge Strength

(c) Example Image (d) Horizontal Edge Strength

Figure 6.1: Examples of the Line Detection Process

In the first image **(a)** there is no dominant peak in the edge strength profile **(b)**. The second image **(c)** does have a definite peak in the edge profile **(d)**. The edge detection software recognises this as a horizontal line in the original image.

Chapter 7

Future Work

The vision system presented here would need many changes to be developed into a system that would be usable in a RoboCup scenario. Additionally there are many changes that could be made to provide incremental improvements to the process either increasing the frame rate or reducing the error.

7.1 Optimisations

Several optimisations can be made to the code that would reduce the time taken to process a frame or the amount of error in the output of the system.

7.1.1 Use of the Field Programmable Gate Array

The most significant change to the system would be moving the colour lookup stage from the SH4 to the board's field programmable gate array (FPGA). Currently the colour lookup process takes about 30 ms to complete (table 6.1). This comprises taking a sample mean of the frame's brightness and performing colour detection on each pixel. The colour detection could be implemented in a ROM as shown in figure 7.1. The brightness (Y) data could be used in the FPGA as well to move the colour detection stage to the FPGA. Colour detection on the FPGA could operate at the same rate that YUV data passes into the FPGA, so each pixel would be analysed and the result sent to the SH4. Not only would this technique reduce the processing by 30ms for a 240×180 image but it would also reduce bus traffic and the amount of memory needed to store the image in the SH4.

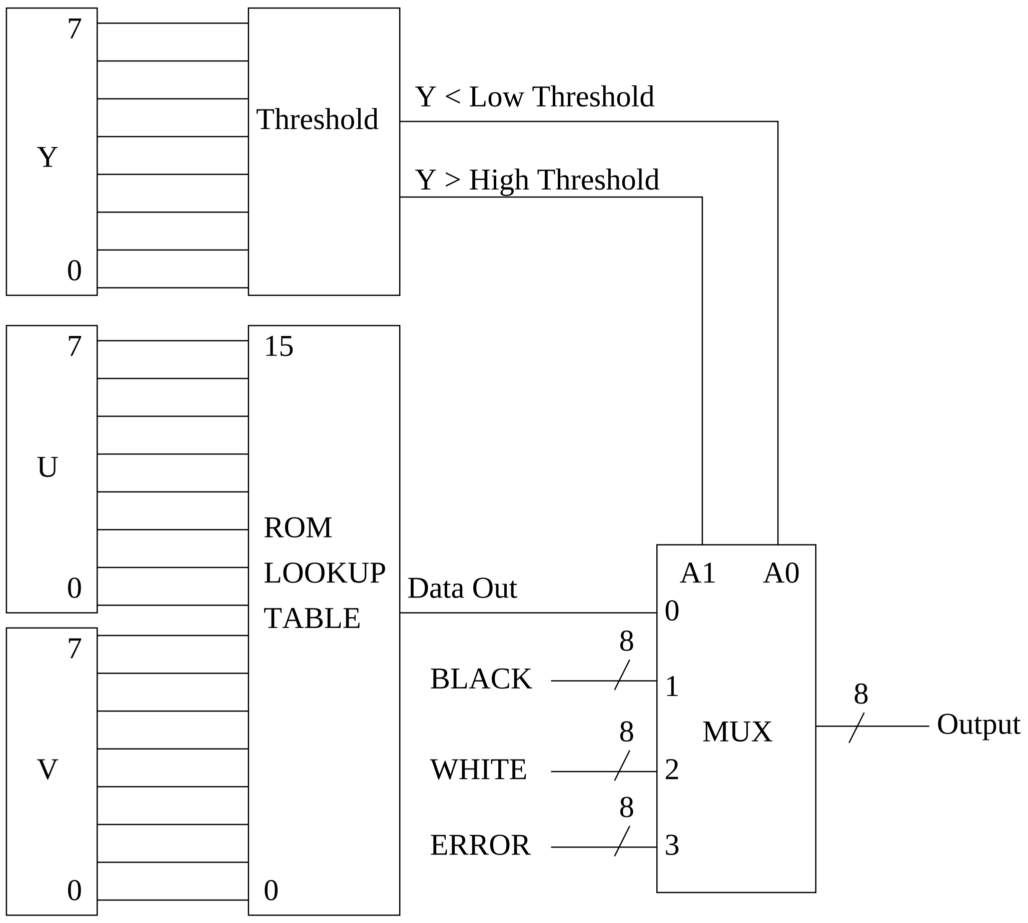


Figure 7.1: Logic Circuit Implementation of Colour Detection.
The U and V values are used as an address into a ROM that contains the values of the lookup table. The thresholded Y value is used to control a multiplexer (MUX) that selects between the lookup table's output and the constant values BLACK and WHITE. The final input to the multiplexer is reserved as ERROR because it should be impossible to have both Y < Low Threshold and Y > High Threshold.

7.1.2 Detection of Green

The reason that green is not detected by the system when working on robot soccer is that it was thought that the large amount of green in each frame would lead to a significantly larger number of run length encoding elements to process in the grouping stage and would lead to no further useful information. However tests show that the effect of adding green detection to the whole colour and blob growing system only increased the frame time by a few milliseconds. The advantage of detecting green is that white pixels with green pixels directly underneath would be good candidates for being part of the edge line.

7.1.3 Faster Morphology

The current morphological system is based on a logical 1×3 erosion of the colour detected image. This erosion uses logical operators but a different solution could be developed. A 1×3 erosion is equivalent to shortening each run length element by one at both ends. This leads to the idea of implementing a complete set of $1 \times n$ morphological operations on the run length encoded data instead of the colour detected image.

Rather than use addition and subtraction to perform these operations it would better to apply them as the run length elements are constructed. For example if an RLE element is only of length one then removing it from the list would be equivalent to an opening of the image with a 1×3 structuring element. A closing could then performed by running the through the list of RLE elements and joining ones that are almost adjacent. This has the potential to reduce the processing time by up to 22 ms.

7.1.4 Noise Rejection

Further work can definitely be done rejecting incorrect matches for red. Some other techniques were considered to remove but never implemented either for reasons of time or apparent complexity.

- With a definite measure of distance from the robot to the ball the approximate size of the ball can be calculated. The difference between the actual and expected sizes gives another measure of error[2]. This technique was not implemented because none of the physical constants such as the height of the robot were available.

- Relating the area of the ball to the bounding rectangle. The trainer program currently displays red blobs with a bounding rectangle and circle with a radius based on the blob area. This gives an indication of the relationship between bounding box and area. Tests show that the relationship is fairly solid and will be suitable for rejecting noise as the relationship noticeably breaks when applied to noisy pixels.
- The final technique considered was to measure the edge length of the blobs and relate this to the area of the pixel. This would require measuring the edge though which makes it less attractive than the previous technique.

7.2 Additions

7.2.1 Coordinate Transforms and iPaq Software

To be useful for robotic motion planning the image coordinates of each object must be transformed into three dimensional coordinates relative to some fixed location. To do this the position and orientation of the camera relative to the robot must be known. For some robots like the ViperRoos this is a fixed position but for more complex robots such as the GuRoo the camera is capable of independent motion. Consequently the vision software must have access to the output of the internal sensors of the robot that report the angles of the limbs and head.

On the GuRoo this information will be passed to the iPaq so it would make sense for the coordinate transform stage of the vision system to be performed here. The two dimensional image coordinates of objects would be transmitted to the iPaq which would then build a transform matrix from the internal sensors information. The matrix would be used to calculate the three dimensional positions of all of the objects.

The PC based trainer is planned to be ported to the iPaq. The trainer would be able to receive the YUV images from the SH4 and use these as test images for training. In addition to this the run length encoded data from the SH4 could be transmitted quickly to the iPaq which could then decode this data and display it on the screen for debugging purposes.

7.2.2 Adapting to the output of the OV7620 Camera

At this point in time the OV7620 camera is not interfaced with the SH4 processor board and the system uses test images taken from a hand held digital camera. These images are converted from RGB to YUV and subsampled using MATLAB. The OV7620 is expected to have at least slightly different characteristics to the trial images. The lookup table would have to be recalibrated and the offsets used for calculating the brightness thresholds would have to be changed.

The use of a real camera would introduce problems of distortion and focusing to the system. Also the availability of a real time data feed would make errors in the system much more apparent than they currently are.

Chapter 8

Conclusion

A basic architecture for colour segmentation has been developed. The system is able to perform fast colour detection using a combination of lookup tables and brightness thresholds, that has been trialled on an embedded processor. The colour detected image has been successfully converted into blobs using a two stage process of run length encoding and grouping.

The YUV colour space is extremely useful for reducing the effect of brightness changes in the image. The other advantage is that YUV allows the use of unchanging lookup tables in U and V which provide an extremely fast way of performing colour detection.

The speed of execution is slightly better than expected (section 3.2). The majority of time is spent performing colour detection, morphological operations and run length encoding. There are many opportunities to increase the speed of the system, including using a field programmable gate array to perform colour detection (section 7.1.1).

The system is capable of detecting most objects although more work does need to be done preventing incorrect matches. Some techniques to correct these problems are outlined in section 7.1.4.

A GUI system has been developed that allows the user to construct a colour lookup table using test images. The performance of the colour detection and later stages of the software can be examined within the GUI environment. This software has also proved useful during the development and debugging of the vision software.

The software is at the stage were it could provide useful information to robot navigating a colour coded environment. However the system needs to be made much more robust before it can operate reliably.

Bibliography

[1] M. Abe, *Building Cross Development Environment Targetting SH4 System*, `www.linuxsh.sourceforge.net/docs/abe/2001320-gcc2.97/README_E.php3`,August, 2001.

[2] T. Bandlow et al, *Fast Image Segmentation, Object Recognition and Localisation in a RoboCup Scenario*, Lecture Notes in Artificial Intelligence Volume 1856, Springer, Berlin, 2000.

[3] R. Bartelds, et al, *Clockwork Orange: The Dutch RoboSoccer Team*, RoboCup 2001, Springer-Verlag, Berlin, 2001.

[4] A. Berry, *Soccer Robots with Local Vision*, Honours Dissertation, Univ. of Western Australia, Dept. of Electrical and Electronic Engineering, 1999.

[5] Blower, A., *Development of a Vision System for a Humanoid Robot*, Undergraduate Thesis, University of Queensland, 2001.

[6] Browning, B., *Robust Vision For Robot Soccer*, Undergraduate Thesis, University of Queensland, 1999.

[7] Bruce, J., *Realtime Machine Vision Perception and Prediction*, Undergraduate Thesis, Carnegie Mellon University, 2000.

[8] Bruce, Balch, Veloso, *Fast and Inexpensive Colour Segmentation for Interactive Robots*, Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2000.

[9] M. Chang, et al, *ViperRoos 2000*, RoboCup 2000: Robot World Cup IV Lecture Notes in Artificial Intelligence 2019, Springer-Verlag, Berlin, 2001.

[10] R. Gonzalez and R. Woods, *Digital Image Processing*, Addison-Wesley, Reading Massachusetts, 1992.

[11] B. Horn, *Robot Vision*, The MIT Press, Cambridge Massachusetts, 1986.

[12] Hosking S, *High Speed Peripheral Interface*, Undergraduate Thesis, University of Queensland, 2001.

[13] Omnivision, *OV7620 Product Specifications - Revision 1.3*, 2000.

[14] W. Pratt, *Digital Image Processing, Second Edition*, John Wiley & Sons, New York, 1991.

[15] D. Paulus and J. Hornegger, *Applied Pattern Recognition: a Practical Introduction to Image and Speech Processing in C++*, Verlag Vieweg, Braunschweig Germany, 1998.

[16] *RoboCup Medium Size League Rules*, `smart..uni-ulm.de/ROBOCUP/f2000/rules01/rules2001.html`, accessed 12 August 2001.

[17] *Robocup Small Size League Rules*, `parrotfish.coral.cs.cmu.edu/robocup-small`, accessed 12 August 2001.

[18] *Robocup Website*, `www.robocup.org`, accessed 12 August 2001.

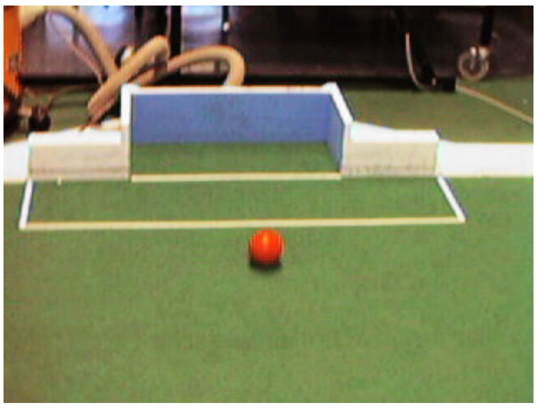
[19] R. Schilling, *Fundamentals of Robotics, Analysis and Control*, Prentice Hall, New Jersey, 1990.

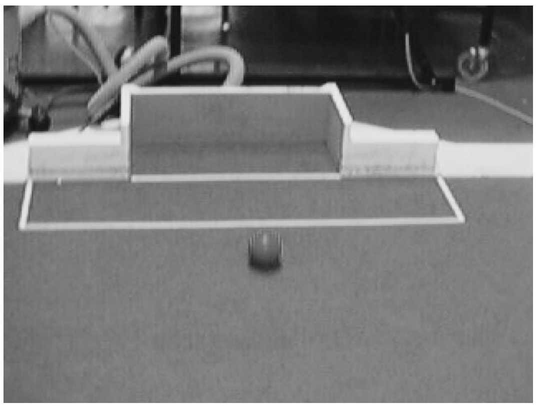
[20] P. Stone, *RoboCup-2000 The Fourth Robotic Soccer World Championships*, AI Magazine, Vol 22, Number 1, Spring 2001.

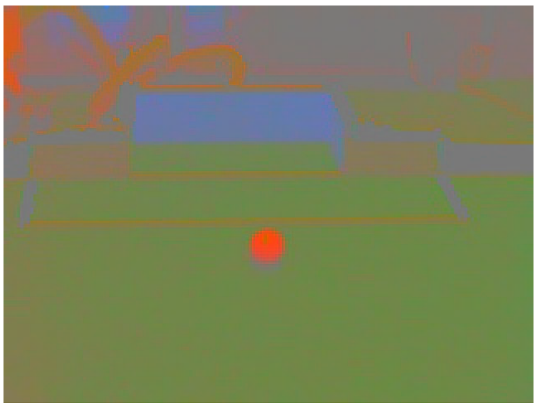
[21] Wagstaff M, *Mechanical Design and Internal Sensors for a Humanoid Robot*, Undergraduate Thesis, University of Queensland, 2001.

Appendix A

Colour Images


(a) RGB Image


(b) Y Comonent


(c) U and V Components

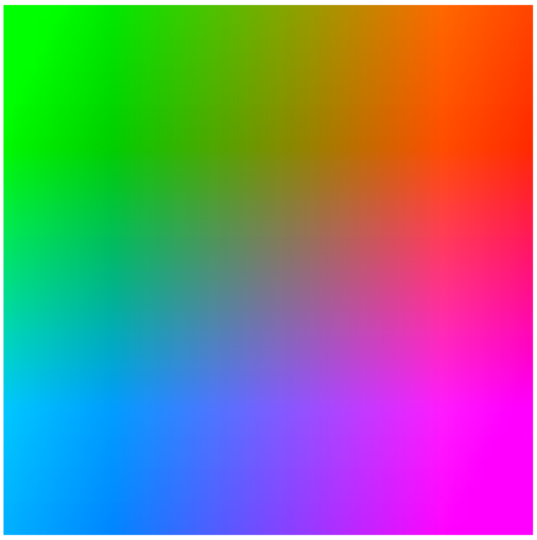
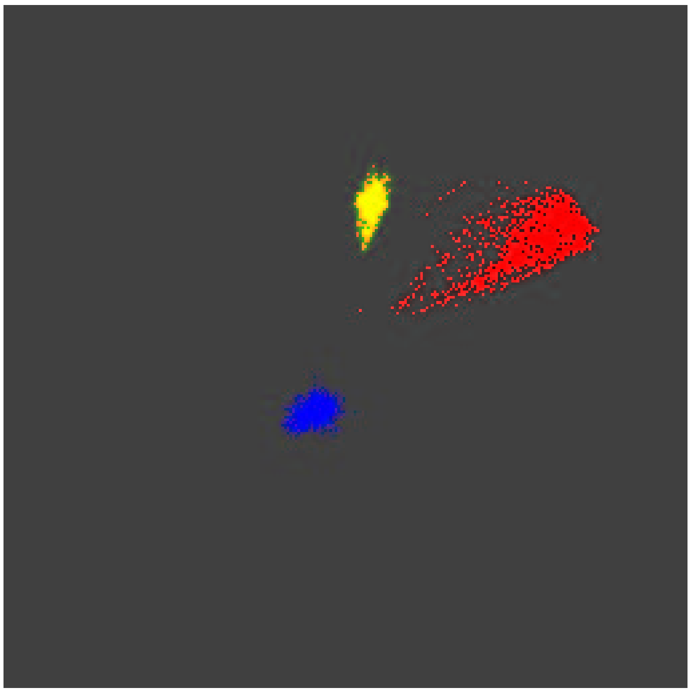
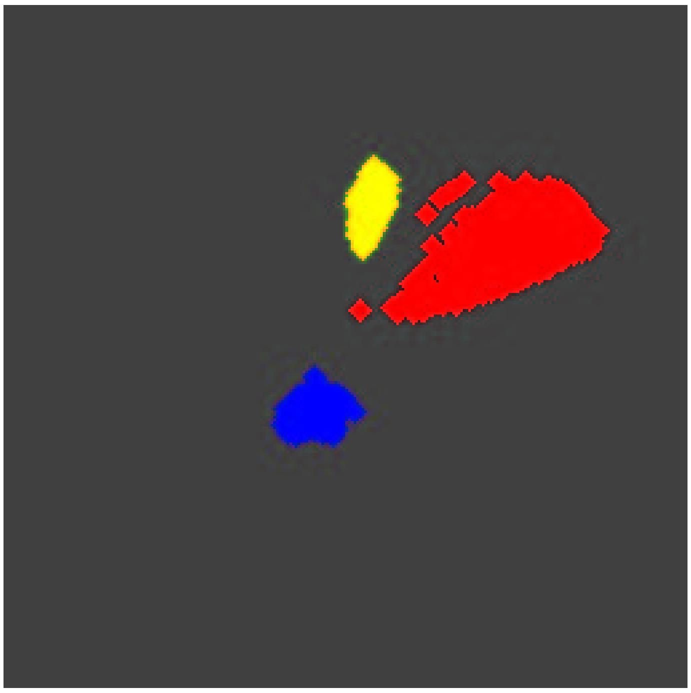

(d) Colour as a function of U and V

Figure A.1: YUV Image Representation
The YUV representation of the image **(a)** can be easily described in two components. The Y component is simply the brightness of each pixel **(b)**. The UV components describe the colour of the pixel **(c)**. The variation of colour with U and V is shown in figure **(d)**. V is the horizontal axis and U is the vertical axis with 0,0 being the top left corner.



(a) Lookup Table Before Dilation



(b) Lookup Table After Dilation

Figure A.2: Trainer Lookup Table
The trainer program uses samples of colours provided by the user to construct the initial lookup table **(a)**. The table is then dilated to fill up the gaps in the image **(b)**.

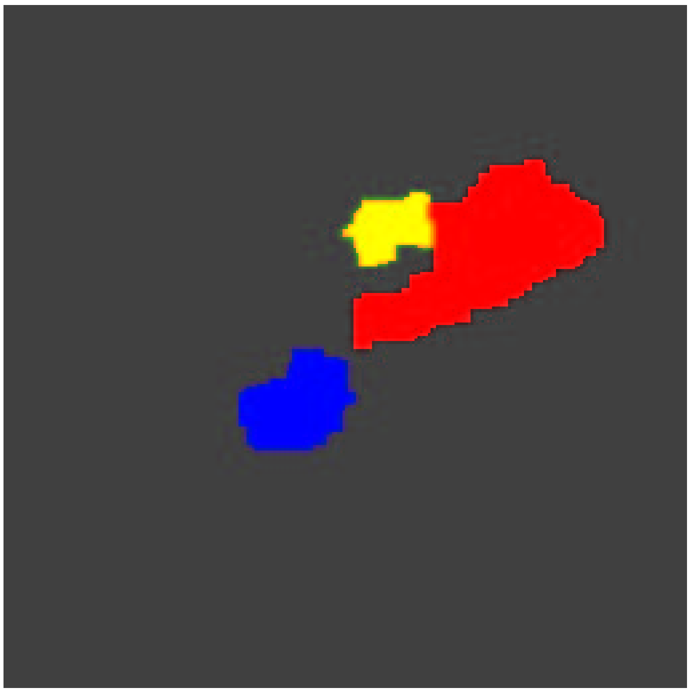
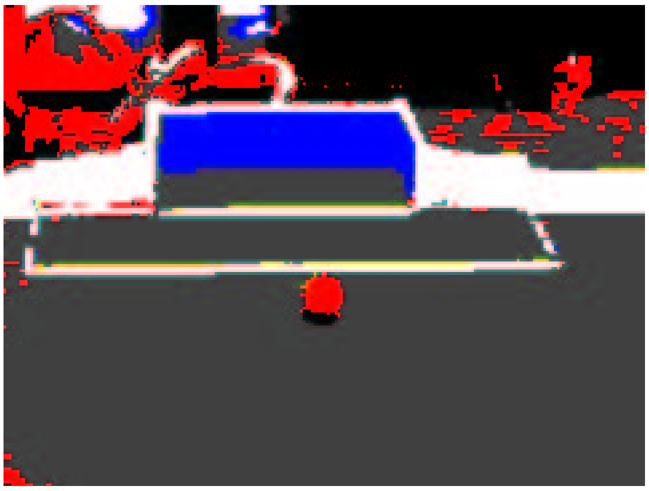
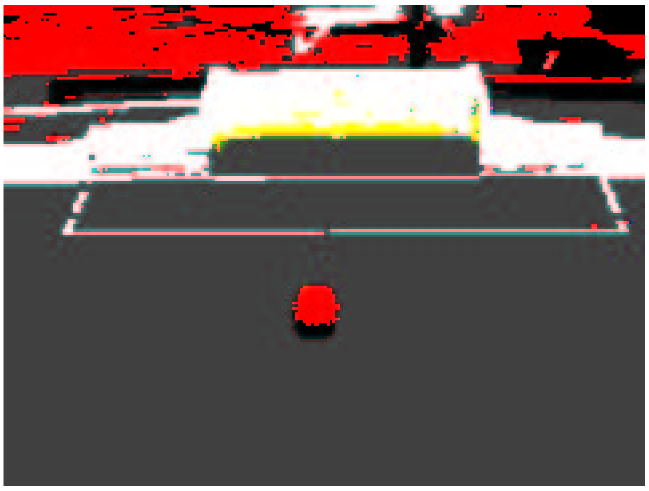


Figure A.3: An example colour lookup table.
The lookup table used for testing, it is configured to detect red, blue and yellow. The horizontal access is V and the vertical access is U with a scale 0 to 255 on both axes.

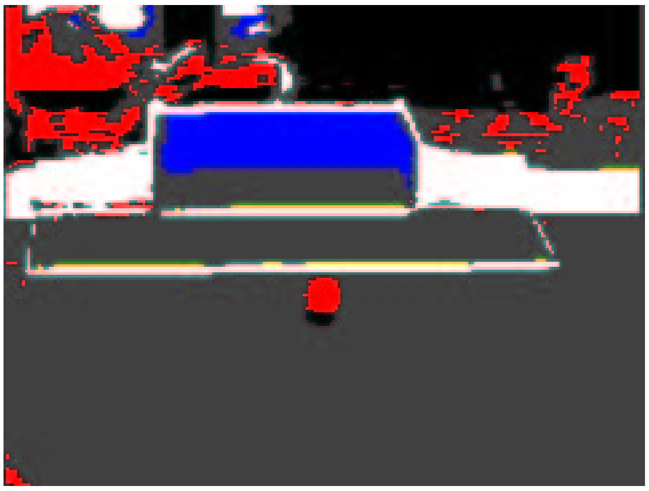


(a) View of Blue Goal



(b) View of Yellow Goal

Figure A.4: Colour Detected Image
The blue, white and black areas are successfully detected. The bright yellow is often interpreted as white.


(a) View of Blue Goal

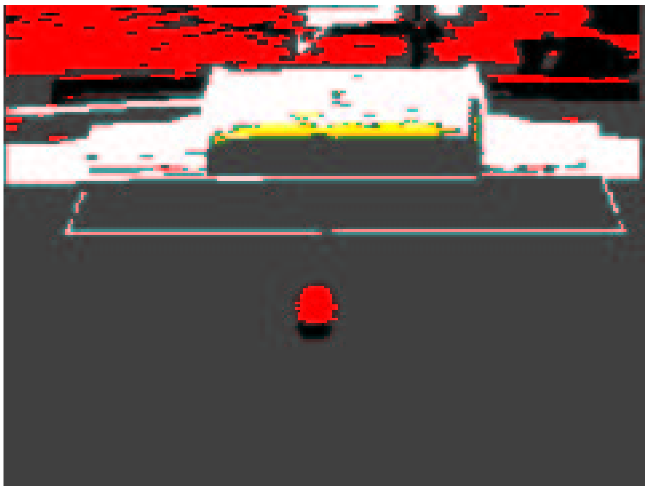
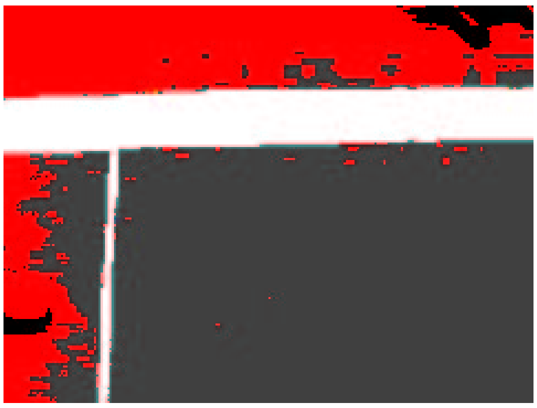
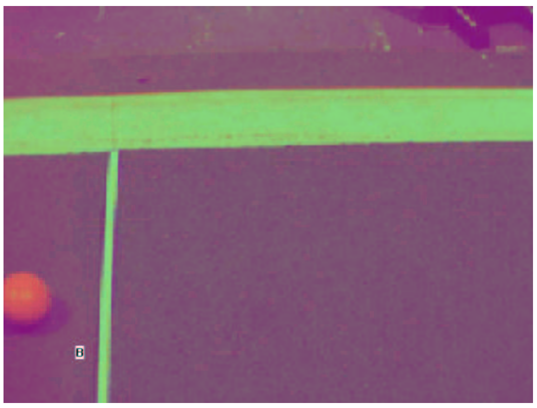

(b) View of Yellow Goal

Figure A.5: Eroded Images
The erosion by a 1×3 mask removes any pixel groups whose dimensions are smaller than 1×3.


(a) Colour Detected


(b) YUV image and Object Locations

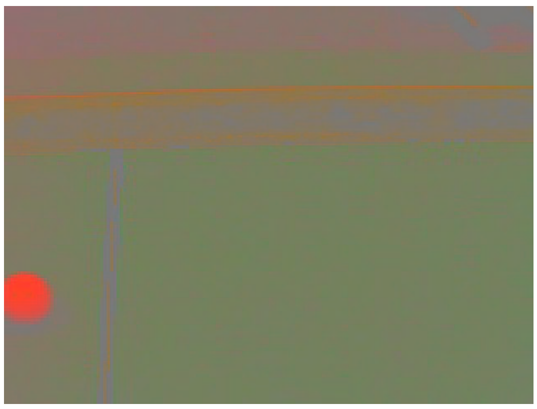
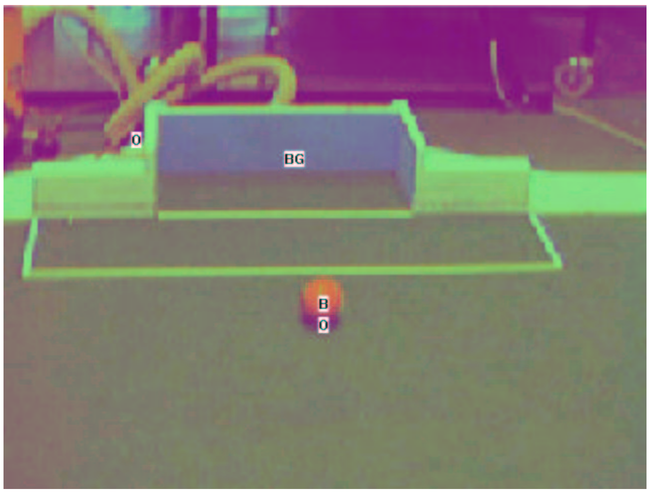


Figure A.6: An Example of Poor Colour Detection
The colour detection fails here because the area of the field on the right of the image is being segmented as red. The brightness normalised version of this image shows that the UV data in this region is borderline between red and green. A better defined lookup table may correct this problem.



(a) View of Blue Goal

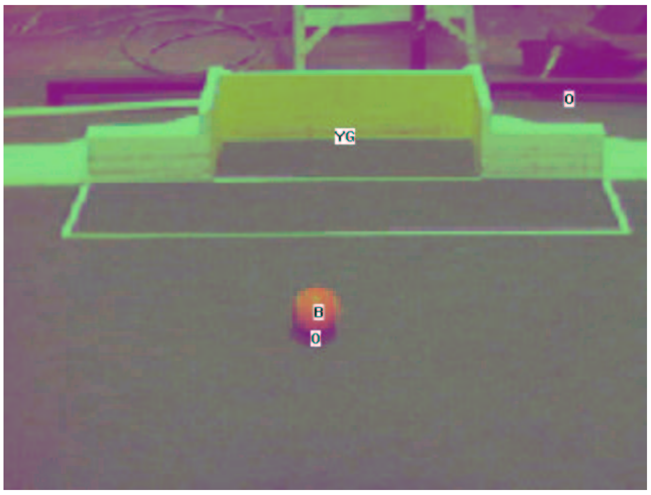


Figure A.7: Detected Objects.
The objects in the image detected by the vision software. The B represents the ball, O represents obstacles(black objects), and the YG and BG represent the yellow and blue goals. The shadow of the ball is interpreted as an obstacle.

Appendix B

Code

The image processing support functions are contained in segment.c and segment.h. The main frame processing function from imtest.c shows the order in which the functions are called.

B.1 Imdemo.c

Program B.1: imtest.c

```c
void imdemo() {
        quick_lookup(SRAMDATA.yuv_image, SRAMDATA.table, SRAMDATA.color_detected);
        //lookup_bytable(SRAMDATA.yuv_image, SRAMDATA.table, SRAMDATA.color_detected);
        erode13(SRAMDATA.color_detected, SRAMDATA.eroded, 240, 180);
        rlecode(SRAMDATA.eroded, SRAMDATA.rle_encoded);
        end_blobs = group(SRAMDATA.rle_encoded, SRAMDATA.blobs);
        end_obstacles = analyse(SRAMDATA.blobs, end_blobs,SRAMDATA.obstacles, &ball,&bluegoal, &yellowgoal);
        edgeprofile(SRAMDATA.eroded, SRAMDATA.profile);
        findedgeline(SRAMDATA.profile);
}
```

B.2 Segment.h

Program B.2: segment.h

```c
#ifndef SEGMENT_H
#define SEGMENT_H

#include "image.h"
//#include <stdio.h>
//#include <stdlib.h>

#ifdef WIN32
        #include <stdio.h>
#else
        #include <sh/gio.h>
#endif

// Memory Allocation
#define ENCODEDSPACE 4096
#define NUMBERBLOBS 512
#define NUMBEROBSTACLES 64

//Image size
```

```
#define TABLEWIDTH 256
#define TABLESIZE (TABLEWIDTH * TABLEWIDTH)

#define IMWIDTH 240
#define IMHEIGHT 180
#define IMSIZE (IMWIDTH*IMHEIGHT)
#define IMBYTESIZE (IMSIZE*3)


//Color Codes
#define FIELDCODE 1
#define WALLCODE 2
#define BALLCODE 4
#define BLACKCODE 8
#define BLUECODE 16
#define YELLOWCODE 32

// Minimum Sizes
#define MINSIZE 6

#define MINWALLSIZE 140
#define MINOBSTACLESIZE 6
#define OBJECTLISTSIZE 10
#define TOPWALLOFFSET 0
#define MINGOALSIZE 10
#define MINEDGELINE 35
#define MINEDGEDIFF (MINEDGELINE/2)

/* Aspect ratio of a square for the chosen
   image dimensions
   (IMWIDTH/640)/(IMHEIGHT/480) */
#define SQUARERATIO 1

#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

//#define USEPRINTF

void find_centroid(image* segmented_image);

struct blobtag {
        //struct blobtag* next;
        //struct blobtag* prev;
        int area;
        int color;
        int xmax,xmin,ymax,ymin;
};

typedef struct blobtag blob;

struct rletag {
        int begin;
        int end;
        int color;
        int tag;
        int y;
        blob* blobpointer;
};
typedef struct rletag rle;
struct objecttag {
        int distance, bearing, color;
};

typedef struct objecttag object;

void lookup_bytable(unsigned int* the_image, UCHAR* table,
                                        UCHAR* out_image);//, int* profile);
blob* group(rle* input, blob* blobs);
void rlecode(UCHAR* segmented_image, rle* output);
void reconstruct(rle* input, image* output);
void color_reconstruct(rle* input, UCHAR* output);

//void dilate22(UCHAR* the_image, UCHAR* out_image);
void code2rgb(UCHAR* input, UCHAR* output, int size);
//object* analyse(blob* blobdata, blob* endblob, object* results,object* pball);
object* analyse(blob* blobdata, blob* endblob, object* results,
                object* pball,object* pygoal, object *pbgoal);

int findedgeline(int* profile);
void erode33(UCHAR* input, UCHAR* output, int width, int height);
void erode13(UCHAR* input, UCHAR* output, int width, int height);

void histogram_region(UCHAR* the_image, int sx, int sy, int ex, int ey, int* hist);
void thresh(int* hist, int code, int level, UCHAR* table);

void edgeprofile(UCHAR* target, int* profile);

#ifdef WIN32
        void quick_lookup(UCHAR* the_image, UCHAR* table, UCHAR* out_image);
        void lookup_bytable(UCHAR* the_image, UCHAR* table,
                                                UCHAR* out_image);
#else
        void quick_lookup(unsigned int* the_image, UCHAR* table, UCHAR* out_image);
        void lookup_bytable(unsigned int* the_image, UCHAR* table,
                                                UCHAR* out_image);
#endif

#endif
/* SEGMENT_H */
```

B.3 Segment.c

Program B.3: segment.c

```
#include "segment.h"
#define WORD long int
#include "angletable.h"
//#include "imageio.h"

#ifndef WIN32
        #include <../vbsh4/test.h>
        #include <sh/sci.h>
#endif

//#include <assert.h>

// y thresholds
unsigned int ytlow, ythigh;


// Sample Mean Constants
#define YMEAN_SAMPLEPOWER 10
#define YMEAN_NUMSAMPLE 1024
#define YMEAN_INCREMENT (IMSIZE/YMEAN_NUMSAMPLE)

#define CD_SUBSAMPLE 4

#ifdef WIN32

        int ymean(UCHAR* the_image) {
                UCHAR* impointer = the_image;
                int ytotal=0;
                int i,y;
                for (i = 0; i < YMEAN_NUMSAMPLE; i++) {
                        y = *impointer;
                        ytotal+=y;
                        impointer += YMEAN_INCREMENT*3;
                }
                return (ytotal >> YMEAN_SAMPLEPOWER);
        };

        #define Y_POSOFFSET (50)
        #define Y_NEGOFFSET (50)

        inline int lookup_pixel(UCHAR* impointer, const UCHAR* table) {
                UCHAR y = *impointer;
                UCHAR u = *(impointer+1);
                UCHAR v = *(impointer+2);
                //unsigned int out;
                if (y  > ythigh) return WALLCODE;
                if (y  < ytlow) return BLACKCODE;
                return table[u*256+v];
        }

        void quick_lookup(UCHAR* the_image, UCHAR* table,
                                        UCHAR* out_image) {
                unsigned int output;
                unsigned int last_output = 0;
                const UCHAR* end_addr = the_image + (IMBYTESIZE);// used to be -subsample
                UCHAR* outpointer = out_image;
                UCHAR* impointer = the_image;
                UCHAR* next_pixel;
        //      UCHAR* end_out = out_image + IMSIZE;
        //      int y,u,v;
                int mean = (ymean(the_image));
                int i;
                ythigh = (mean) + Y_POSOFFSET;
                ytlow = (mean) - Y_NEGOFFSET;
                *outpointer = lookup_pixel(impointer, table);
                while (impointer < end_addr) {
                        next_pixel = impointer+CD_SUBSAMPLE*3;
                        output = lookup_pixel(next_pixel, table);
                        if (output == last_output) {
                                for (i = 0; i < CD_SUBSAMPLE; i++) {
                                        *(++outpointer) = output;
                                }
                                //printf("s");
                                impointer= next_pixel;
                        } else {
                                for (i = 0; i < CD_SUBSAMPLE-1; i++) {
                                        impointer+=3;
                                        outpointer++;
                                        *outpointer = lookup_pixel(impointer, table);
                                };
                                // then do the last one which has already been looked up
                                outpointer++;
                                impointer= next_pixel;
                                *(outpointer) = output;
                                last_output = output;
                        }
                        //assert(outpointer < end_out);
                };
        };


#else
        #define Y_POSOFFSET (50<<16)
        #define Y_NEGOFFSET (50<<16)

        int ymean(unsigned int* the_image) {
                unsigned int* impointer = the_image;     int ytotal=0;
```

```
		int i,y;
		for (i = 0; i < YMEAN_NUMSAMPLE; i++) {
			y = ((*impointer)>>16)&0xFF;
			ytotal+=y;
			impointer += YMEAN_INCREMENT;
		}
		return (ytotal >> YMEAN_SAMPLEPOWER);
	};

	inline int lookup_pixel(unsigned int* impointer, const UCHAR* table) {
		unsigned int data = *impointer;
		unsigned int y = (data);
		//unsigned int out;
		if (y  > ythigh) return WALLCODE;
		if (y  < ytlow) return BLACKCODE;
		return table[data & 0x0000FFFF];
	}



	void quick_lookup(unsigned int* the_image, UCHAR* table,
				UCHAR* out_image) {
		unsigned int output;
		unsigned int last_output = 0;
		const unsigned int* end_addr = the_image + (IMSIZE);//used to be -subsample
		UCHAR* outpointer = out_image;
		unsigned int* impointer = the_image;
		int* next_pixel;
	//	UCHAR* end_out = out_image + IMSIZE;
	//	int y,u,v;
		int mean = (ymean(the_image))<<16;
		int i;
		ythigh = (mean) + Y_POSOFFSET;
		ytlow = (mean) - Y_NEGOFFSET;
		*outpointer = lookup_pixel(impointer, table);
		while (impointer < end_addr) {
			next_pixel = impointer+CD_SUBSAMPLE;
			output = lookup_pixel(next_pixel, table);
			if (output == last_output) {
				for (i = 0; i < CD_SUBSAMPLE; i++) {
					*(++outpointer) = output;
				}
				//printf("s");
				impointer= next_pixel;
			} else {
				for (i = 0; i < CD_SUBSAMPLE-1; i++) {
					impointer++;
					outpointer++;
					*outpointer = lookup_pixel(impointer, table);
				};
				// then do the last one which has already been looked up
				outpointer++;
				impointer= next_pixel;
				*(outpointer) = output;
				last_output = output;
			}
			//assert(outpointer < end_out);
		};
	};


// This code is slower than the quick lookup version by about 4 fps at 240*180
void lookup_bytable(unsigned int* the_image, UCHAR* table,
			UCHAR* out_image) { //, int* profile) {
	const unsigned int* end_addr = the_image + IMWIDTH * IMHEIGHT;
	UCHAR* outpointer = out_image;
	unsigned int* impointer = the_image;
	unsigned int mean = ymean(the_image)<<16;
	ythigh = mean + Y_POSOFFSET;
	ytlow = mean - Y_NEGOFFSET;
	//int i;
	while (impointer < end_addr) {
		*outpointer = lookup_pixel(impointer++, table);
		outpointer++;
	};
};


#endif

// Slow Erosion
/*void erode13(UCHAR* input, UCHAR* output, int width, int height) {
	register int x,y;
	register UCHAR* inpointer = input + 1;
	register UCHAR* outpointer = output + 1;
	UCHAR* end_addr = input+IMSIZE -1;
	while (inpointer < end_addr) {
		*outpointer = *(inpointer-1) & *(inpointer) & *(inpointer+1);
		outpointer++;
		inpointer++;
	}
}*/


//erode13 only works for a specific image width
#if IMWIDTH != 240
	#error "Incorrect width"
#endif


void erode13(UCHAR* input, UCHAR* output, int width, int height) {
	int x,y;
	UCHAR* inpointer = input;
	UCHAR* outpointer = output;
```

```
        unsigned int a,b,c;

        for (y = 0; y < height; y ++) {
                *outpointer++ = 0;
                inpointer++;
                a = *(inpointer - 1);
                b = *(inpointer);
                c = *(inpointer + 1);

                for (x = 1; x < width - 4; x+=3) {
                        *outpointer++ = a & b & c;
                        inpointer++;

                        a = *(inpointer+1);
                        *outpointer++ = b & c & a;
                        inpointer++;

                        b = *(inpointer+1);
                        *outpointer++ = c & a & b;
                        inpointer++;

                        c = *(inpointer+1);

                }
                //inpointer and outpointer now have finished column 178
                //and still point to it.
                *outpointer++ = 0;
                *outpointer++ = 0;
                inpointer+=2;

        }
}

void rlecode(UCHAR* segmented_image, rle* output) {
// watch for runs that end at the rightmost column
register UCHAR* impointer = segmented_image;
rle* rlepointer = output;
#ifndef NDEBUG
        rle* spacecheck = output + ENCODEDSPACE;
#endif
int tagcounter = 1;
register int x,y;
register int data;
int rowcount;
int color, oldend;
//int height = segmented_image->height;
//int width = segmented_image->width;
int inrun = 0;
for (y = 0; y < IMHEIGHT; y ++) {
        inrun = 0;
        rowcount = 0;
        oldend = 0;
        for (x = 0; x < IMWIDTH; x ++) {
                data = *impointer;
                if (inrun  && (data != color)) {
                        rlepointer->end = x-1;
                        oldend = rlepointer->color;
                        #ifdef USEPRINTF
                                printf("[tag: %d b: %d ",rlepointer->tag,rlepointer->begin);
                                        printf("e: %d c: %d]\n",rlepointer->end, rlepointer->color);
                                #endif
                                rlepointer++;
                                inrun = 0;

                                #ifdef USEPRINTF
                                        rowcount++;
                                        printf("rowcount %d\n", rowcount);
                                #endif
                        }

                        //logic this line
                        if (!!data && !inrun) {
                                rlepointer->begin = x;
                                color = data;
                                rlepointer->color=color;
                                rlepointer->tag = tagcounter++;
                                rlepointer->blobpointer = 0;
                                rlepointer->y = y;
                                inrun = 1;
                                #ifdef USEPRINTF
                                        printf("{Start Run x:%d y:%d}", x, y);
                                #endif
                        };

                        impointer++;

                        //assert(rlepointer < spacecheck);
                        //assert(rlepointer >= output);
                };
                if (inrun) {
                        rlepointer->end = IMWIDTH - 1;
                        #ifdef USEPRINTF
                                printf("[tag: %d b: %d ",rlepointer->tag,rlepointer->begin);
                                printf("e: %d c: %d]\n",rlepointer->end, rlepointer->color);
                        #endif
                        rlepointer++;
                        //Set the next one to zero
                        inrun = 0;
                }
        };
        rlepointer->tag= 0;   //null terminate the run array
}

// Converts the rle objects back to colour codes
```

```
void reconstruct(rle* input, image* output)
{
        int x,y;
        UCHAR* data = output->data;
        rle* rlepointer = input;

        //for each row of the image
        y = 0;
    //for all the rle runs on this row
        while(rlepointer->tag) {
                for(x = rlepointer->begin; x < rlepointer->end + 1; x++) {
                        *(data + x + output->width*rlepointer->y) = rlepointer->tag;
                }
                rlepointer++;
        };

};

//Colour codes to RGB
void code2rgb(UCHAR* input, UCHAR* output, int size)
{
        UCHAR* impointer = input;
        UCHAR* out_pointer = output;
        const UCHAR* end_addr = input + size;
        UCHAR r,g,b;
        while(impointer <  end_addr) {
                switch(*(impointer++)) {
                        case FIELDCODE:
                                        r = 0;          g = 255;        b = 0;
                                break;
                        case WALLCODE:
                                        r = 255;        g = 255;        b = 255;
                                break;
                        case BALLCODE:
                                        r = 255;        g = 0;          b = 0;
                                break;
                        case BLACKCODE:
                                        r = 0;          g = 0;          b = 0;
                                break;
                        case BLUECODE:
                                        r = 0;          g = 0;          b = 255;
                                break;
                        case YELLOWCODE:
                                        r = 255;        g = 255;        b = 0;
                                break;
                        default:
                                        r =  64;        g = 64;         b = 64;
                                break;
                }
                #ifdef WIN32
                        *(out_pointer++) = b;
                        *(out_pointer++) = g;
                        *(out_pointer++) = r;
                #else
                        //assert(0);
                        *(out_pointer++) = r;
                        *(out_pointer++) = g;
                        *(out_pointer++) = b;
                #endif



        };
};

#ifdef WIN32
//FIX optimize
void color_reconstruct(rle* input, UCHAR* output)
{
        int x,y;
        UCHAR r,g,b;
        UCHAR* data = output;
        UCHAR* impointer;
        rle* rlepointer = input;



        for (impointer=data;impointer<data+IMSIZE;impointer++) *impointer = 96;
    //for all the rle runs on this row
        while(rlepointer->tag) {
                y = rlepointer->y;
                switch(rlepointer->color) {
                        case FIELDCODE:
                                        r = 0;  g = 255;        b = 0;
                                break;
                        case WALLCODE:
                                        r = 255;g = 255;        b = 255;
                                break;
                        case BALLCODE:
                                        r = 255;g = 0;          b = 0;
                                break;
                        case BLACKCODE:
                                        r = 0;  g = 0;          b = 0;
                                break;
                        case BLUECODE:
                                        r = 0;  g = 0;          b = 255;
                                break;
                        case YELLOWCODE:
                                        r = 255;g = 255;        b = 0;
                                break;
                }
                for(x = rlepointer->begin; x < rlepointer->end + 1; x++) {
                        #ifdef WIN32
                                *(data + 3 * x + 3 * IMWIDTH * y) = b;
                                *(data + 3 * x + 3 * IMWIDTH * y + 1) = g;
                                *(data + 3 * x + 3 * IMWIDTH * y + 2) = r;
                        #else
                                *(data + 3 * x + 3 * IMWIDTH * y) = r;
                                *(data + 3 * x + 3 * IMWIDTH * y + 1) = g;
                                *(data + 3 * x + 3 * IMWIDTH * y + 2) = b;
                        #endif

                }
                rlepointer++;
        };
};
#endif

// Sub part of group
inline rle* matchrows(rle* rlepointer, rle* nextrow, blob** nextblob,int y ) {
    rle* rowbelow = nextrow;
    int begin = rlepointer->begin; //optimize
    int end = rlepointer->end; //optimize
    int color = rlepointer->color;
    blob* belowblob;
    int abovearea;
    blob* aboveblob = rlepointer->blobpointer;
    int belowarea;

    while(rowbelow->y==y+1) {
        //Now check the for connectivity.
        #ifdef USEPRINTF
                printf("$%X{%d,%d",rowbelow->tag,rowbelow->y,rowbelow->begin);
                printf("-%d} ",rowbelow->end);
        #endif
        if ((((rowbelow->begin >= begin)&&(rowbelow->begin <= end))||
            ((rowbelow->end >= begin) && (rowbelow->end <= end))||
                ((rowbelow->begin <= begin) && (rowbelow->end >= end))) &&
                (rowbelow->color == color)) {
                if (rowbelow->tag > rlepointer->tag) {
                    //Subsume below into above
                    //Create a blob for the above
                    if (!aboveblob) {
                        rlepointer->blobpointer = (*nextblob)++;
                        aboveblob = rlepointer->blobpointer;
                                aboveblob->color = rlepointer->color;
                                abovearea = end - begin + 1;
                                    aboveblob->area = abovearea;
                                    aboveblob->ymax = y;
                                    aboveblob->ymin = y;
                                    aboveblob->xmax = end;
                                    aboveblob->xmin = begin;
                    }

                    if(rowbelow->blobpointer) {
                                                //assert(aboveblob);
                        //row below is a blob get info
                        belowblob = rowbelow->blobpointer;
                                                //this test could be better done with tag numbers
                                                if (aboveblob != belowblob) {
                                                        aboveblob->area += belowblob->area;
                                                        belowblob->area = 0;
                                                };
                        aboveblob->ymax = MAX(aboveblob->ymax, belowblob->ymax);
                        aboveblob->ymin = MIN(aboveblob->ymin, belowblob->ymin);
                        aboveblob->xmax = MAX(aboveblob->xmax, belowblob->xmax);
                        aboveblob->xmin = MIN(aboveblob->xmin, belowblob->xmin);
                        //assert(belowblob);
                        rowbelow->blobpointer=rlepointer->blobpointer;
                        rowbelow->tag = rlepointer->tag;
                    } else {
                                                //assert(aboveblob);
                        //just a row so calc stats
                        belowarea = rowbelow->end-rowbelow->begin + 1;
                        aboveblob->area += belowarea;
                        //aboveblob->ymin not need updating?
                        //assert(aboveblob->ymin<y+1);
                        aboveblob->ymax = MAX(aboveblob->ymax, y+1); //unlikely
                        aboveblob->xmin = MIN(aboveblob->xmin, rowbelow->begin);
                        aboveblob->xmax = MAX(aboveblob->xmax, rowbelow->end);
                        rowbelow->blobpointer=aboveblob;
                        rowbelow->tag = rlepointer->tag;
                    };
                } else {
                    //subsume above blob into below blob.
                    belowblob = rowbelow->blobpointer;
                    if (aboveblob) {
                                                // rlepointer has a blob
                                                if (aboveblob != belowblob) {
                                                        belowblob->area += aboveblob->area;
                                                        aboveblob->area = 0;
                                                }

                                                belowblob->ymin=MIN(belowblob->ymin,aboveblob->ymin);
                                                belowblob->ymax=MAX(belowblob->ymax,aboveblob->ymax);
                                                belowblob->xmin=MIN(belowblob->xmin,aboveblob->xmin);
                                                belowblob->xmax=MAX(belowblob->xmax,aboveblob->xmax);

                                                //assert(aboveblob);
                                                rlepointer->blobpointer = rowbelow->blobpointer;
                                        } else {
                                                //rlepointer just a single row
                                                //assert(belowblob);
                                                abovearea = end - begin + 1;
                                                belowblob->area+=abovearea;
                                                belowblob->ymin=MIN(belowblob->ymin,y);
                                                belowblob->ymax=MAX(belowblob->ymax,y);
                                                belowblob->xmin=MIN(belowblob->xmin,begin);
                        belowblob->xmax=MAX(belowblob->xmax,end);
                        rlepointer->blobpointer = rowbelow->blobpointer;
                    };
                    rlepointer->tag   = rowbelow->tag;
        }
        #ifdef USEPRINTF
            printf("m$%X ",rowbelow->tag);
          #endif
        }
        rowbelow++;//and now is here
      }
      return rowbelow+1;  //Should be pointing to the start of the next row;
}



blob* group(rle* input, blob* blobs)
{
    int y = input->y;
    rle* rlepointer = input;
    rle* rowbelow;
    rle* nextrow = input;
    rle* temprow = nextrow;
    blob* nextblob = blobs;
    int begin, end;
    blob* spacecheck = blobs+NUMBERBLOBS;
    while((++temprow)->y==y) {};

    //for each row of the image
    while(y < IMHEIGHT-1) {   // HEIGHT MINUS ONE DON'T DO LAST ROW
        rlepointer=nextrow;
        y = rlepointer->y;
        //  now find next row after this oneif it is not y++
        //  then skip this row too
        while((++nextrow)->y==y) {}; //could be optimized
        //nextrow = temprow;
        if (nextrow->y == y+1) {
          //for all the rle runs on this row
          while(rlepointer < nextrow) {
            #ifdef USEPRINTF
                begin = rlepointer->begin;
                end = rlepointer->end;
                printf("Tag $%X{y %d - ",rlepointer->tag,rlepointer->y);
                printf("%d %d}  [",begin,end);
            #endif
            //rowbelow = input + ENCODEDWIDTH * (y+1);
            rowbelow = nextrow;
            temprow = matchrows(rlepointer, nextrow, &nextblob, y );
            //assert(nextblob<spacecheck);
        //call match here returns last rle on the row
              //next run along the current row
              #ifdef USEPRINTF
                printf("]\n");
              #endif
              rlepointer++;
              if (rlepointer->tag == 0) return nextblob;
        }
      };
  };
  #ifdef USEPRINTF
    printf("END GROUP\n");
  #endif
    return nextblob;
}

inline int xcentroid(blob* target) {
    return (target->xmin + target->xmax)>>1;
};

inline int ycentroid(blob* target) {
    return (target->ymin + target->ymax)>>1;
};

#define MAX_ERROR 4
object* analyse(blob* blobdata, blob* endblob, object* results,
        object* pball,object* pygoal, object *pbgoal)
{
    blob* pblob = blobdata;
    blob* wall = 0;
    blob* ball = 0;
    blob* bgoal = 0;
    blob* ygoal = 0;
    int bluesize = MINGOALSIZE;
    int yellowsize = MINGOALSIZE;
    object* nextresult = results;
    int expected_height,error;

    int topline = 0;
    int wallsize = MINWALLSIZE;
    int ballsize = 0;

    for (pblob = blobdata; pblob < endblob; pblob++) {
        switch (pblob->color) {
            case WALLCODE:
                if (pblob->area > wallsize) {
                    wallsize = pblob->area;
                    wall = pblob;
                };
            break;
        }
    }

    if (wall) {
        topline = wall->ymin - TOPWALLOFFSET;

        #ifdef USEPRINTF
            printf("walldimension %d x %d y - ",wall->xmin, wall->ymin);
            printf(" %d x %d y\n", wall->xmax, wall->ymax);
            printf("horizon line %d\n",topline);
            printf("top of interest %d\n",topline);
        #endif
    };
//  topline = 0;
    for (pblob = blobdata; pblob < endblob; pblob++)
        if ((pblob->area)&&(pblob->ymin>topline)) {
            #ifdef USEPRINTF
                printf("%d %d ", pblob->color, pblob->area);
                printf("%d %d ", pblob->xmin, pblob->ymin);
                printf("%d %d \n", pblob->xmax, pblob->ymax);
            #endif


            switch (pblob->color) {
                case BALLCODE:
                    expected_height = (pblob->xmax-pblob->xmin+1)*SQUARERATIO;
                    error = expected_height - (pblob->ymax - pblob->ymin +1);
                    if  ((error < 6) && (error > -3)) {
                        if (pblob->area > ballsize) {
                            ballsize = pblob->area;
                            ball = pblob;
                        };
                    }
                break;
                case BLACKCODE:
                    if (pblob->area > MINOBSTACLESIZE) {

                        nextresult->distance =ycentroid(pblob);
                        nextresult->bearing = xcentroid(pblob);
                        nextresult->color = BLACKCODE;
                        nextresult++;
                    };
                break;
                case BLUECODE:
                    if (pblob->area > bluesize) {
                        bgoal = pblob;
                        bluesize = pblob->area;
                    };
                    break;
                case YELLOWCODE:
                    if (pblob->area > yellowsize) {
                        ygoal = pblob;
                        yellowsize = pblob->area;
                    };
                    break;
            }
        };
    if (ball) {
        //pball->distance = angletable[ycentroid(ball)];
        pball->distance = ycentroid(ball);
        pball->bearing = xcentroid(ball);
        pball->color = BALLCODE;
    } else {
        pball->distance = 0;
        pball->bearing = -40;
        pball->color = 0;
    };
    if (bgoal) {
        pbgoal->distance = ycentroid(bgoal);
        pbgoal->bearing = xcentroid(bgoal);
        pball->color = BLUECODE;
    } else {
        pbgoal->distance =0;
        pbgoal->bearing = -40;
        pball->color = 0;
    }
    if (ygoal) {
        pygoal->distance = ycentroid(ygoal);
        pygoal->bearing = xcentroid(ygoal);
        pygoal->color = YELLOWCODE;
    } else {
        pygoal->distance =0;
        pygoal->bearing = -40;
        pygoal->color = 0;
    }

    return (nextresult);
};


int findedgeline(int* profile) {
    int y;
    int sum;
    int max=MINEDGELINE;
    int oldmax = 0;
    int besty =0;
    for (y=0;y< IMHEIGHT; y++) {
        sum = profile[y];
        if(sum > oldmax) {
            if (sum > max) {
                oldmax = max;
                max = sum;
                besty = y;
            } else {
                oldmax = sum;
            };
        }
    };
    if (max - oldmax > MINEDGEDIFF)
```

```
                return besty;
        else
                return 0;
};

/*void dilate22(UCHAR* the_image, UCHAR* out_image) {
        UCHAR mask;
        int x,y;

        for (y = 1; y<IMHEIGHT - 1; y++)
                for (x = 0; x<IMWIDTH - 1; x++) {
                        mask = *(the_image+x+IMWIDTH*y);
                        if (mask != 0) {
                                *(out_image+(x+1)+IMWIDTH*y) = mask;
                                *(out_image+(x+2)+IMWIDTH*y) = mask;
                                *(out_image+(x+3)+IMWIDTH*y) = mask;
                        };
        }
}*/


//
// uchar for histogram
void histogram_region(UCHAR* the_image, int sx, int sy, int ex, int ey, int* hist) {
        int x, y;
        UCHAR u,v;
        for (y = sy; y <= ey; y++)
                for (x = sx; x <= ex; x++) {
                        u = *(the_image + 3 * x + y * IMWIDTH * 3 + 1);
                        v = *(the_image + 3 * x + y * IMWIDTH * 3 + 2);
                        (*(hist + v + TABLEWIDTH * u))++;
                }
}

// Threshold the the histogram into a UCHAR lookup table
void thresh(int* hist, int code, int level, UCHAR* table) {
        int x, y;
                for (x = 0; x < TABLEWIDTH; x++)
                        for (y = 0; y < TABLEWIDTH; y++)
                                if (*(hist + TABLEWIDTH * y + x) > level)
                                        *(table + TABLEWIDTH * y + x) = code;
};


void edgeprofile(UCHAR* target, int* profile) {
        int y,x;
        UCHAR* impointer = target;
        for (y = 0; y < IMHEIGHT - 1; y++)
                for (x = 0; x < IMWIDTH; x++) {
                        profile[y]+= (*impointer==WALLCODE) && (*(impointer+IMWIDTH) != WALLCODE);
                        impointer++;
                }
}
```