Recognising a Pattern-Featured Object in a RoboCup Environment

Michael Green

15th August 2003

Contents

1 Introduction **5**
- 1.1 Problem Definition 5

2 RoboCup: The Robot World Cup Soccer Games **7**
- 2.1 Sony Legged Robot League 8
- 2.2 Challenges 9
- 2.3 Team Sweden 9

3 The AIBO Robot **11**
- 3.1 Overview of AIBO 11
- 3.2 Hardware and Peripherals 11
- 3.3 The Aperios Operating System 13
- 3.4 The OPEN-R Software Development Kit 13

4 Team Sweden Software Architecture **15**
- 4.1 Manipulation and Walking 15
- 4.2 Perception 16
- 4.3 Behaviours 17
- 4.4 Localisation 17
- 4.5 Image Processing and Object Recognition 17

5 The Problem **19**
- 5.1 Overview of Challenge 1 19
- 5.2 Properties of a Ball 19
- 5.3 Perceiving the Ball 20
- 5.4 Related Work 21

6 Solving the Ball Recognition Problem **23**
- 6.1 Computer Vision and Image Analysis 23
  - 6.1.1 The YCbCr Colour Space 23
  - 6.1.2 Image Analysis 24

6.1.3 Image Processing Techniques 24
6.2 Technical Preparations 25
6.3 Using a Grey-scale Image 26
6.3.1 Defining and Using the Roberts Cross 27
6.3.2 Analysing the Histogram 28
6.3.3 The Creation of Blobs 30
6.3.4 Recognising the Ball 31
6.4 Using a Colour Image 32
6.4.1 Long Range Algorithm 33
6.4.2 Close Range Algorithm 34
6.5 Results and Implementation 35

7 Conclusions and Future Improvements 39
7.1 Gray-scale Approach 39
7.2 Colour Approach 39
7.3 Challenge 1 40
7.4 Suggested Improvements 40

A Terminology 45
A.1 Abbreviations 45

B Installation and Configuration of OPEN-R SDK 47
B.1 Background 47
B.2 Solution 47
B.2.1 Installing the Development Tools 48
B.2.2 Installing the OPEN-R SDK 48
B.2.3 Installing the Sample Programs 49
B.2.4 Mounting the MSAC-US1 Device 49
B.3 Results 50

C C++ Code for the Gray-scale Approach 51

D C++ Code for the Colour Approach 76

Chapter 1

Introduction

1.1 Problem Definition

RoboCup [1] is an international research and education initiative to foster AI and robotics in general. The aim of RoboCup is to be able to beat the human world champions in soccer, using humanoid robots, by the year 2050. The Sony Legged Robot League(SLRL) [2] is one of many leagues present in RoboCup, using a soccer game as the standard problem. This specific league uses one of Sony's own robots, namely AIBO [3]. Naturally the SLRL, as a part of the RoboCup initiative, has to constantly evolve and improve the performance of the soccer game. Evolution in this sense is to remove every bit of information that a human does not need in order to play a game of soccer, e.g., landmarks, separate goal colours, etc. Since RoboCup is also starting to attract public attention it is very important to have a good game play, which means that trying new in the actual soccer game environment would be a poor solution. Thus every year there are three technical challenges defined, concerning topics that needs to be added to, or removed from, the soccer game. This year one of the topics concerns using a black and white soccer ball instead of the current orange one. The recognition of the black and white soccer ball is the first of the three challenges, hence it will be referred to as Challenge 1 throughout this report. In conclusion, the problem addressed here will consist of the following stages:

1. Investigate the possibility of recognising a black and white ball using the AIBO robot.

2. Develop an algorithm that can be used in Challenge 1.

3. Test the performance of the algorithm and the theories behind it in Challenge 1.

Chapter 2

RoboCup: The Robot World Cup Soccer Games

RoboCup is an international effort to create a standard problem for a wide range of AI robotic technologies, so that they can be integrated and evaluated. Originally it was called "Robot World Cup Initiative". As a standard domain RoboCup chose a soccer game. With this in mind they stated an aim for the initiative:

> By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, complying with the official rules of the FIFA, against the winner of the most recent World Cup.

This competition is intended for a team of multiple fast-moving robots under a dynamic environment. Several aspects of AI have to be integrated in such a team in order to successfully participate in RoboCup. They include: design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning, robotics, and sensor fusion.

There are three major domains of RoboCup, each serving a special purpose. The RoboCupJunior consists of a soccer challenge, a dance challenge and a rescue challenge. It is primarily intended for young students competing with simple robots. Then there is the RoboCupRescue which is based on two parts: Rescue Simulation League and Rescue Robot League. This is an effort to develop efficient search and rescue robotics in the field of large scale disasters. It is a rather new branch of RoboCup and demands heterogeneous agents that can perform activities like long range planning and emergent collaboration. There is also a non-trivial logistic factor in this branch. The final part of RoboCup is the RoboCupSoccer which contains five different leagues: Simulation League, Small Size Robot League, Middle Size Robot League, Sony Legged Robot League and Humanoid League. This report is only concerned with the Sony Legged Robot League so we will only discuss that part of RoboCup in the remaining part of this section.

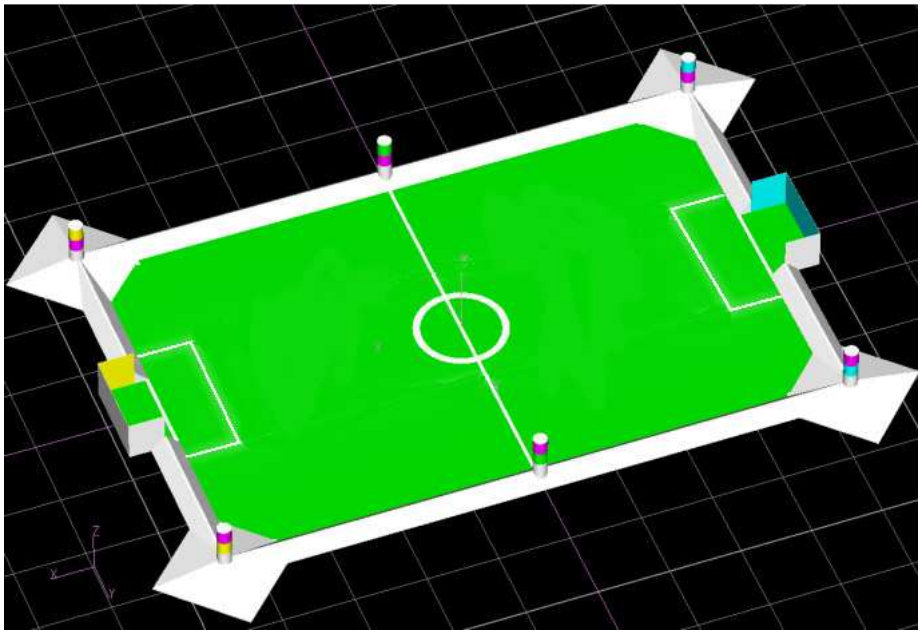

Figure 2.1: The field used in Sony Legged Robot League.

2.1 Sony Legged Robot League

In the Sony Legged Robot League, abbreviated SLRL, teams of autonomous AIBO's are competing against each other in a soccer game. This takes place in a well defined soccer field which is illustrated in Figure 2.1. There are six landmarks on the field that the players may use for localisation. It also has different colours of the goals to further help a player determine its position and orientation.

A soccer game consists of three parts equally divided over 30 minutes. The first half of the game, a break, and the second half of the game. Each of the teams has four members including a goalkeeper. During the game there are many rules that have to be obeyed. These rules basically comply with the rules of a human soccer game, i.e., one ball, two teams, two goals, two goalkeepers, etc. However, an AIBO can hardly play the game in the same way a human can, so naturally there have to be other restrictions regarding kicking and handling of the ball. For example; a SLRL player may only hold the ball for three seconds, which means that it has to kick or dribble the ball within this time frame. A goalkeeper, on the other hand, may hold the ball for up to five seconds as long as it is in the penalty area. Another important rule is that only the goalkeeper may stay in this area. Players breaking this rule are transferred back to the half-way line. The rule is applicable even if the goalkeeper is outside the penalty area. Moreover players

may not by physical contact obstruct each other from reaching the ball.

2.2 Challenges

The soccer game is the biggest part of SLRL but there are also three different challenges for the teams to participate in as well. Each of these challenges targets a specific problem area within AI robotics, such as vision, localisation, cooperation. The challenges in 2003 were:

1. Black and White Ball: The purpose of this challenge was to evaluate AIBO's ability to recognise a black and white soccer ball and to score a goal with it.

2. Localisation: This challenge was supposed to evaluate localisation ability in the absence of explicit markers such as landmarks.

3. Obstacle Avoidance: Avoiding unnecessary collisions with other robots will be demanded in the future, so this challenge was meant to investigate AIBO's ability to avoid obstacles.

The sole purpose of these challenges is to take RoboCup to the next level, which means that all efforts to reduce the difference between real soccer and RoboCup soccer will be tried in a challenge before they are included in the game.

2.3 Team Sweden

Team Sweden [7] is the Swedish national team that participates in RoboCup's SLRL since 1999. The team was created in 1998 and involved three universities in Sweden located in Stockholm, Ronneby and Örebro. Despite the distance between these cities the team was able to create a successful project organisation and cooperation. Team Sweden entered the SLRL with the following corner-stones in mind:

Scientific value: The software should illustrate its scientific approach to autonomous robotics, and demonstrate its research lines in this field.

Generality: The software should embody general principles that are needed to achieve autonomous robot operation, and that can be reused in different robots operating in different environments.

Effectiveness: The software should effectively address the specific challenges present in the RoboCup domain in general, and in the legged robot league in particular.

Robustness: The software should degrade smoothly in face of errors and imprecision in perception and execution; in particular, the lower layers should still provide some reasonable response even when the higher layers can not compute a reliable course of action.

Team Sweden plans to entered the SLRL held in Padova, Italy 2003. Currently the team involves four academic sites: Lund University, Blekinge Institute of Technology, the University of Örebro and the University of Murcia, Spain. Each of these sites has been given an area of responsibility. For instance, Lund University has been assigned Challenges 1 and 2, i.e., the vision and localisation parts of the challenges.

Chapter 3

The AIBO Robot

In this section the AIBO robot will be introduced, starting by describing its current hardware. Later a brief overview of Sony's real-time operating system Aperios will be given. A quick description of the OPEN-R Software Development Kit concludes this chapter.

3.1 Overview of AIBO

There are several different models of AIBO on the market today which means that AIBO is actually a collective name for a series of robots. At the moment there are 4 different models of AIBO, ERS-210, ERS-220, ERS-311 and ERS-312. This document will only refer to the ERS-210 model since that is the only model currently used in the Sony legged league. A picture of the ERS-210 model can be found in Figure 3.1.

3.2 Hardware and Peripherals

According to the hardware documentation [6], AIBO has a number of sensors, which correspond to the senses of humans and animals - touch, hearing, sight and a sense of balance. The core of AIBO is a 64 bits RISC processor operating at 384 MHz, using a MIPS architecture. There is also an IEEE 802.11b wireless interface that allows for wireless communication equipped. A number of sensors are present in the robot. These sensors, and other hardware in AIBO, can be categorised in the following manner:

Touch sensors sensitive to pressure are located on AIBO's head, chin, back and legs. These sensors can register pressure between 0.0N and 2.94N.

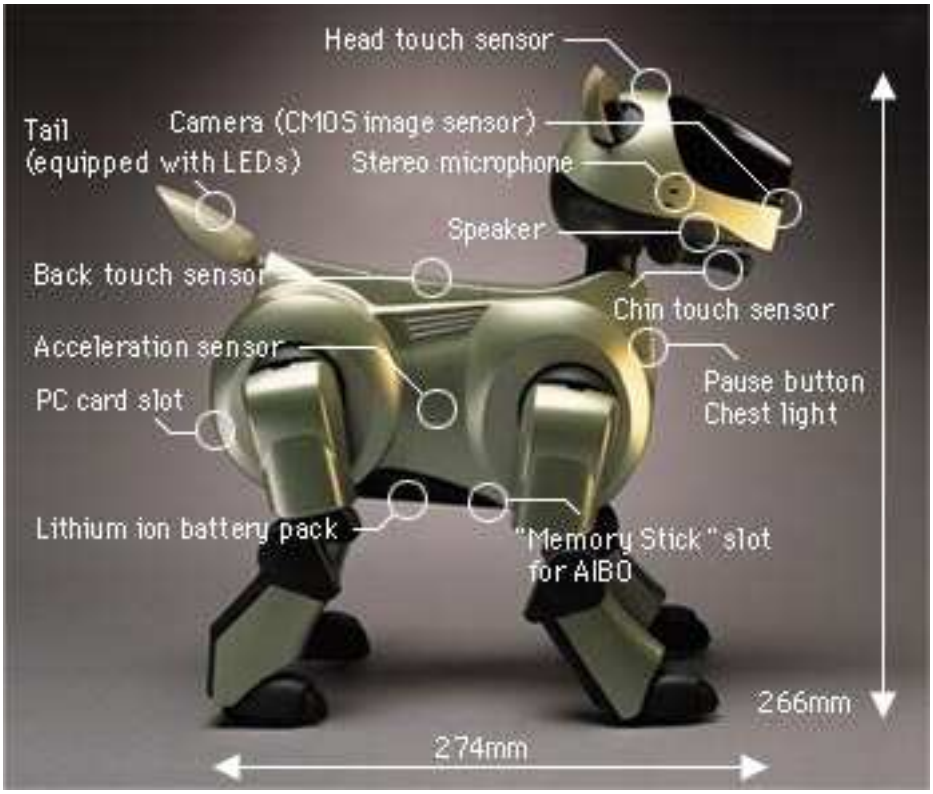

Figure 3.1: The ERS-210 model of AIBO.

Vision is accomplished by a CMOS image sensor capable of taking pictures at 25FPS. The maximum resolution measured in pixels is 352x288. The camera has a 57.6 horizontal angle of view. Vertically it only has 47.8 degrees angle. AIBO has also a distance sensor operating at ranges between 10 and 90cm.

Hearing can be accomplished by utilising the microphones located at each side of AIBO's head. The sound can be recorded in stereo at a sampling frequency of 16000Hz in 16bits linear PCM.

Audio Output from the speaker located in AIBO's "mouth" can only be done at 8000Hz in 8bits linear PCM. Since there is only one speaker the sound is limited to mono playback.

Balance is achieved by using the internal acceleration sensor. The sensor is sensitive to movement in all three dimensions, i.e., left-right, front-back, up-down. The measuring range is between -2G and 2G.

AIBO has 9 movable parts: four legs, a head, two ears, a chin and a tail. Some of these parts have several joints which allows for a wide range of movements. Mobility of a robot can be measured in “degrees of freedom”, i.e., the number of single motions it can perform. For example, an automatic phonograph turntable has three degrees of freedom. It can spin the turntable, it can raise and lower the stylus arm, and it can move the arm laterally to the first track. Due to the many joints of AIBO it has about 20 degrees of freedom.

3.3 The Aperios Operating System

Aperios is an object-oriented, distributed real-time operating system. It was formerly known as Apertos [14] and exists in many of Sony’s products today, including AIBO. It is very small compared to other operating systems as it only takes up about 100 KB of storage.

Aperios is based upon a reflective architecture which allows the software to monitor its own functions and reconfigure itself on the fly as demand on it changes. This is accomplished by letting objects be the only constituent of the operating system and allow them to migrate between different environments. Usually an object defines its own properties and semantics, but in order to maintain object heterogeneity Aperios separates this task from the object itself and creates a set of metaobjects. These metaobjects are responsible for supplying the object with properties and a set of metaoperations, i.e., the semantics of the object. There are several metaobjects and they are ordered in a metahierarchy. An object in Aperios is in other words nothing more than a container of information that may roam freely in a distributed environment.

Using a Java analogy, a set of metaobjects of an object can be viewed as a virtual machine for that object. It is merely a way of supplying the object with a set of abstract instructions. Of course in order to actually allow the object to run on different platforms a metacore must always be present. A metacore is also a metaobject which provides all other objects with common primitives. It can be compared to a micro-kernel.

3.4 The OPEN-R Software Development Kit

OPEN-R is a software development kit written by Sony Corporation. It serves as an abstraction layer to the Aperios operating system and includes a compiler, binary utilities, software libraries and sample programs. The compiler used with OPEN-R is developed from GCC[1]. OPEN-R also includes a feature called *Remote*

[1]GCC is a C/C++ compiler written and maintained by GNU.

Processing which gives the programmer a possibility to execute parts of the program on a different platform, e.g., a computer. It also allows limited debugging and testing of programs without AIBO.

OPEN-R is a layered architecture based on two layers: a *System layer* and an *Application layer*. The system layer takes care of the actual communication with the hardware of AIBO. It also provides the application layer with a set of services and an interface to the TCP/IP protocol stack. The services provided are:

- input/output of sound data;
- input of image data;
- output of control data to joints;
- input of data from various sensors.

OPEN-R is a modularised and object-oriented software where each module is called an object. OPEN-R objects are substantially different from a standard C++ object. There are similarities though. An easy way of considering a module is to augment the capabilities of the C++ object. Each module is easily replaceable, and allows multiple entry points. It can also communicate with other modules in the program through inter-object communication. Every OPEN-R program is built on one or several concurrently running modules. The fact that the system is modularised makes it easy to replace a module without recompiling the whole program. This works because there is no direct communication between modules in OPEN-R. As we shall see next, all communication is defined externally.

Inter-object communication is the way modules pass information between each other in OPEN-R. It is achieved by using message passing through predefined communication channels. The channels are defined in a configuration file named CONNECT.CFG which is loaded at boot time. In module communication there is always a sending module and a receiving module, called *subject* and *observer* respectively. The subject sends a NotifyEvent to the observer and waits for a reply containing a ReadyEvent. This event serves to inform the subject whether it is OK to send the data or not. In other words data is only sent if the observer is ready to receive it. Every time a transfer of data has been made the observer must once again inform the subject about its current state by sending a ReadyEvent.

A module can of course have any number of subjects and observers implemented but it can only process one message at a time. The reason for this is that every module in OPEN-R is single-threaded. Thus every module must have its own message queue in order to process every message sent to it.

Chapter 4

Team Sweden Software Architecture

Team Sweden has developed a simple and straightforward architecture for its software, inspired by the Thinking Cap [4] architecture. It is built on three fundamental layers: lower, middle and upper layer. Each of these layers has its own set of tasks. The lower layer is responsible for sending motor commands to AIBO, and to some extent react on sensor input. It also incorporates different walking styles [12]. The middle layer is supposed to keep a consistent local model of the world and also define a set of simple behaviours that can be combined and put in a hierarchy. The upper layer handles real-time planning and strategies, and tries to maintain a global map of the environment.

Each layer in this architecture has a set of modules that perform the tasks defined above. As it can be seen in Figure 4.1, there are six modules altogether, and they all handle their own specific tasks. We will look at some important tasks and describe their execution in the modules.

4.1 Manipulation and Walking

In the lower layer there is one module at work called Commander(CMD). It is responsible for sending motor commands to AIBO and generating interrupts when needed, for instance, when someone picks up the robot and moves it. In order to provide the middle layer with a simple interface for manipulating AIBO, the CMD implements head commands, motion commands and some manipulation commands. This module actually represents the entire lower layer since there are no other modules resident in this layer.

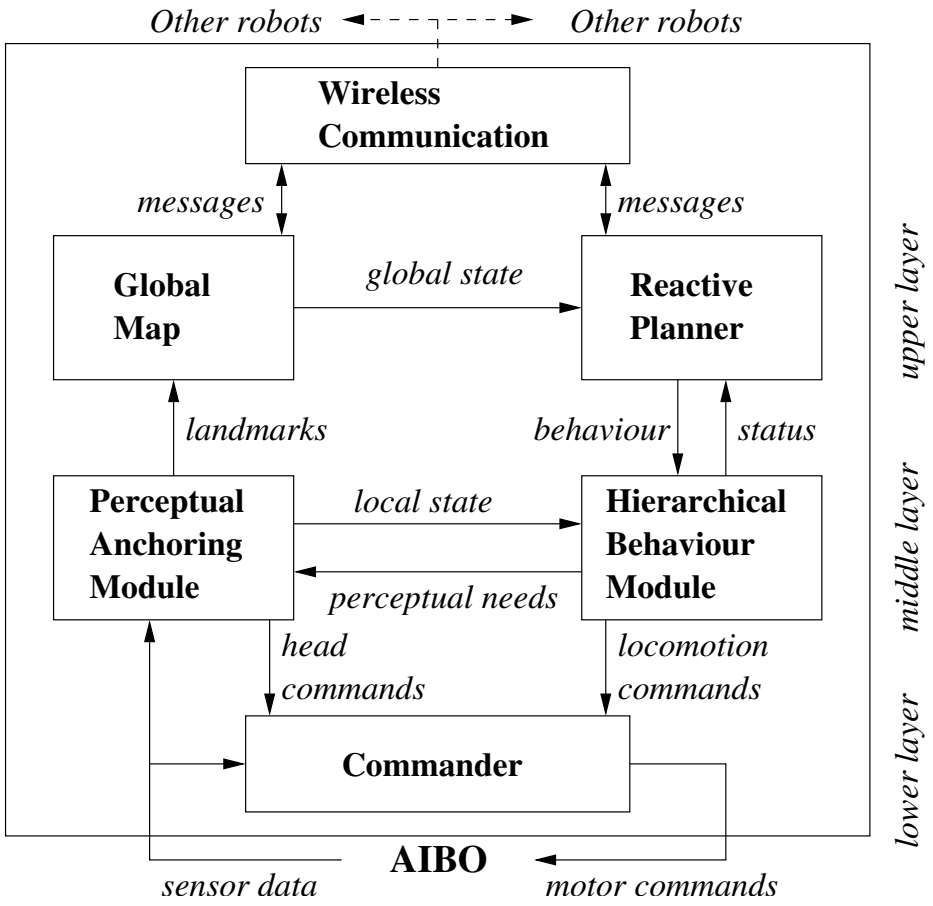

Figure 4.1: The Team Sweden software architecture.

4.2 Perception

All perceptual information is processed in the Perceptual Anchoring Module(PAM) [9], before it is sent further up the hierarchy. One of its major tasks is to identify objects, e.g., players, nets, and landmarks, in the environment by using a colour segmentation. More about this will be said in section 4.5. The recognised objects are positioned in a consistent and updated model of the robots surroundings. To succeed it uses the following three approaches:

perceptual anchoring which means that it updates the position of an object each time it is detected by the camera;

global information is used for the static objects and only when the robot relocates itself;

odometry lets the PAM calculate the position of an object in the local model in relation to AIBO's movement.

A local state is then transferred to the Hierarchical Behaviour Module(HBM). Another task laid upon the PAM is to recognise landmarks in the field and to send information about them to the Global Map(GM) for further processing.

4.3 Behaviours

The Hierarchical Behaviour Module(HBM) [10] defines a set of basic motion behaviours that can be combined into larger ones using meta-rules and hierarchical structures. This way the HBM incorporates several complex behaviours that the Reactive Planner(RP) can use for developing its strategies. At this point the strategies are merely generated from a set of actions based on the electric field approach(EFA) [11]. This approach is actually an extended variant of a potential field which means that a set of positive and negative charges are distributed to the nets and to the players. A heuristic value of a certain configuration is obtained by calculating the charge at the position of the ball. The configuration that yields the best heuristic value is chosen and the RP generates a plan that hopefully will get AIBO there. The RP then sends this plan to the HBM as a behaviour.

4.4 Localisation

Localisation is implemented in the Global Map(GM) module which utilises a fuzzy gridmap to estimate AIBO's position. It relies on the PAM to provide it with information about the landmarks and the nets. However, since the camera is needed for many other things than keeping track of landmarks(e.g., following the ball), the GM can only receive this type of information sporadically. By combining these observations with the robot's own movement an approximate gridmap can be derived. All of this information is combined into a global state, which is in turn sent to the RP.

4.5 Image Processing and Object Recognition

An image, or more specifically a digital image, is often represented in a computer as a 2D array. Each element consists of three values describing the colour of the element, i.e., the pixel. The camera in AIBO uses the YCbCr colour system to represent these three values. More about the YCbCr colour space will be said in

Section 6.1.1. The recognition of different objects in the SLRL environment is accomplished in four steps:

- Segmenting the image into different colours, creating blobs.
- Applying a region growing algorithm to the newly created blobs.
- Merging all blobs of the same colour that are close to each other.
- Classifying each blob by its colour and size.

The colour segmentation is done entirely by the hardware in AIBO and OPEN-R provides methods for retrieving the results of that segmentation. Team Sweden uses this hardware segmentation mainly as a means for producing seeds [13] for their seeded region-growing(SRG) algorithm [8]. Basically, this algorithm considers the difference in colour between the pixels surrounding the seed and the seed itself. The difference is defined via a threshold in YCbCr colour space that is incremental, i.e., the threshold lets the algorithm know how much two adjacent pixels may differ in order to be classified as the same colour. The pixels surrounding the seed, whose colours differ less than the threshold will be included in the seed[1].

The SRG algorithm often performs rather well. However, problems like sharp shadows often creates gaps between the pixels belonging to the object. For this reason Team Sweden uses a blob-merging algorithm that searches for blobs of similar colour and evaluates the distance. If the distance is small enough, e.g., one or two pixels, the blobs are merged. This way the final classification of the blobs will be more accurate since it depends only on the colour and the size.

[1] It actually stops being a seed as soon as the region has grown. It is then referred to as a blob.

Chapter 5

The Problem

5.1 Overview of Challenge 1

As mentioned earlier the basic problem in Challenge 1 lies in recognising a miniature version of a football. Challenge 1 features a RoboCup environment that uses a closed world assumption, i.e., every object is fully described. Thus an agent can have as much *a priori* information as it wants about every encountered object. There are however disturbances in this environment, such as human referees or lighting fluctuations. Moreover, this environment is inaccessible [23] due to the fact that an agent, such as AIBO, can never see the entire environment at once using its sensory system. The environment is also nondeterministic and nonepisodic since there are several agents present. To further complicate matters the environment is also dynamic and continuous which means that the agent has to scan the environment both before and during deliberation. In other words AIBO has to be able to deliberate while performing an action.

Going back to the ball recognition problem, we find a number of difficulties to overcome, e.g., What are the properties of the ball? Are they perceivable? How can they be represented? What is the best way to filter out false positives?, etc. All of these questions has to be answered in such a manner that they are robust with respect to the environment issues discussed above.

5.2 Properties of a Ball

An obvious feature is the shape and colour of the patches. Since the black patches are pentagons and the white patches are hexagons it would be convenient to check that this property holds for every black and white patch. However, due to the fact that 176x144 pixels is the highest resolution that OPEN-R can provide, it is hardly feasible to check the shape of a single patch, even if the ball is very close.

The patch approach could still hold though if one disregards the shape condition. Even though the quality of the image is rather limited, the camera is able to capture white and black patches. These patches will from now on be referred to as blobs. This brings us to the conclusion that a ball will consist of a number of white and black blobs, which is convenient since this condition will always hold. Of course it is only applicable within a certain range due to the fact that a ball that is only a couple of centimetres away from the camera will appear to have only one big white patch. The effect is the same, if not worse, when the ball is far away since the black and white patches will blend and create one grayish blob. This difficulty will be addressed in Section 6.4.1.

5.3 Perceiving the Ball

A colour segmentation, which is already implemented in the hardware of AIBO, could be used. There are however many setbacks to deal with while using this approach. A major problem with colour segmentation on AIBO is that the image is represented in YCbCr colour space, which is a very poor choice when dealing with colour classification. Another downside of this is that the hardware uses a rather low-resolution image to do its segmentation on, i.e., 88x72 pixels. This means that the black patches will not be visible even when the camera is close to the ball. However, the idea might still work if combined with a region growing algorithm. In fact this is how Team Sweden recognises all of the objects present in the RoboCup environment. But these objects are fairly large and one-coloured which makes the creation of usable colour tables possible. This is not the case with the black and white ball since it consists of two colours that are inherently difficult to create colour tables for. A simple explanation to this is that black is more an absence of a colour than a colour, and white is just a mixture of all colours. White also easily adapts to surrounding colours which means that near the ground the white colour looks greenish and by the border it looks very white. The black and white patches will also blend to a certain degree when captured by the camera which also complicates this approach. Another way to acquire the blobs would be to use a convolution mask[1] of some sort that could enhance strong gradients in the picture. Typically we would like to use an edge detection operator, e.g., Roberts Cross [18], Sobel [17] or Canny [19] operators. As it will be explained in the following sections the convolution masks are efficient and easy to use, but they demand a tremendous amount of CPU cycles to do their work, so they are not really optimal when it comes to real-time applications.

[1] See Section 6.1.3

5.4 Related Work

Since no RoboCup prior to 2003 has been using a black and white ball there was not much information to retrieve from the earlier proceedings. Most work regarding object recognition regards large one-coloured objects, e.g., landmarks, nets, etc. Comparing the orange ball used in the soccer game with the black and white ball used in the challenge we find three distinct differences:

- The orange ball is composed of only one colour;
- Orange does not absorb colours to the same extent as white does;
- Orange does not appear in any other object in the RoboCup environment.

Despite these differences J. Bruce, T. Balch, and M. Veloso proposed an interesting and general approach to segmentation and thresholding [22].

Chapter 6

Solving the Ball Recognition Problem

In this section two fundamentally different approaches to solving the ball recognition problem are proposed. A discussion of their strengths and weaknesses follows. In the first approach a gray-scale image was used, i.e., only the Y component of the YCbCr image received from OPEN-R. The second approach uses a colour image.

6.1 Computer Vision and Image Analysis

Computer vision is a huge topic, and has been constantly developing since the 1950s. It is also closely related to AI since many of todays robots depend on input from different types of cameras.

6.1.1 The YCbCr Colour Space

According to Webster's Revised Unabridged Dictionary (1913) the term colour is defined as:

> A property depending on the relations of light to the eye, by which individual and specific differences in the hues and tints of objects are apprehended in vision; as, gay colours; sad colours, etc.

Digital images are always presented in colour, whether it is one or many colours, and there are numerous theories regarding how to represent them. In other words a colour space is simply a means for defining colours. One of the most common colour space is the RGB system which is based on the theory of human vision, i.e., it defines all colours as different amounts of red, green, and blue.

The camera in AIBO uses the YCbCr colour space, which is also known as YUV. It is also divided into three components. Y is luminance. Cb(U) and Cr(V) are different chrominance components. Cb represents the chrominance of blue, and Cr the chrominance from red. An RGB to YCbCr conversion is presented below.

$$Y = Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16 \tag{6.1}$$

$$Cb = U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128 \tag{6.2}$$

$$Cr = V = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128 \tag{6.3}$$

This system is more difficult to comprehend than RGB mainly due to the fact that the human eye does not classify colours in this manner. However, due to the fact that the neurons located in the human eye are divided into colour sensitive(cones) and non-colour sensitive(rods), the components in the YCbCr system makes sense. As it can be seen in the equations above the YCbCr system is also defined using the three fundamental colours. In fact this is true for many colour spaces, since the cones are sensitive to red, green, and blue.

6.1.2 Image Analysis

When analysing an image it is vital to sort out non-important properties, e.g., noise or colours that are of no interest for the problem. Image Analysis can be divided into two stages:

- Image processing;
- Image analysis.

Image processing relates to the preparation of an image for later analysis and use. An image is rarely in a condition that can be used directly by image analysis routines. It may require some preparatory manipulation, e.g., filtering, enhancement, segmentation, etc. In other words; image processing is the collection of routines and techniques that improve, simplify, enhance, or otherwise alter an image. Image analysis is the collection of processes in which a captured image that is prepared by image processing is analysed in order to extract vital information about the image and to identify objects or facts about the object or its environment.

6.1.3 Image Processing Techniques

As previously mentioned, image processing incorporates several techniques for altering an image. One of the most common technique is the convolution mask,

which can be applied to many different types of problems, e.g., edge detection and various filtering. A convolution mask is a $N * N$ matrix containing different patterns of numbers depending on the wanted effect on the image. The mask alters the image by shifting itself over the image and multiplying its value with the corresponding pixel values of the image. A convolution mask can be applied to an image according to equation (6.4), where I is the original image, M is the convolution matrix and I' is the new image created from the old one.

$$I'_{y,x} = \frac{\sum_{i=0}^{n} \sum_{j=0}^{n} M_{i,j} * I_{y+i,x+j}}{S'} \tag{6.4}$$

$$S' = \begin{cases} S, & S \geq 1 \\ 1, & S = 0 \end{cases} \tag{6.5}$$

$$S = \left| \sum_{i=0}^{n} \sum_{j=0}^{n} M_{i,j} \right| \tag{6.6}$$

Another widely used technique is segmentation, which divides the image into different colours by using intensity thresholds [20]. For example, an image with a number of shades of yellow should have two thresholds defining an interval. All pixel intensities belonging to that interval will then be classified as yellow. This method is useful when dealing with environments that have predetermined colours. A soda can would be ideal to recognise using segmentation since they often have a colour that separates them from the rest of the environment.

Thresholding is in itself a technique for filtering information in gray-scale images. It is often combined with a histogram [21] of the image in order to decide upon a threshold. A histogram is a table describing the frequency of pixels at every intensity. A histogram for the image presented in Figure 6.1 can be seen in Figure 6.3. Histograms are not bound to image processing as they can also be used during the analysis of images. After thresholding an image it often ends up as a binary image, i.e., only featuring two colours, black and white. There are many binary operations, e.g., dilation, erosion, skeletonization, thickening, etc., available and the explanation of them are beyond the scope of this thesis.

6.2 Technical Preparations

The first thing that had to be done was to enable all the hardware to work with Linux, since all of the previous thesis work was done using Windows. Among other hardware, the Sony MemoryStick Reader/Writer and the Airport from Apple had to be configured. Also the OPEN-R SDK needed to be installed in order to develop applications for AIBO. Moreover the AIBO robots that were at hand

also needed a BIOS update. More detailed information about the OPEN-R SDK installation and hardware configuration can be found in Appendix B.

6.3 Using a Grey-scale Image

Using only the Y component of the YCbCr image retrieved from AIBO relies on the assumption that the white and black colours are greatly separated in luminance or, more common, intensity[1]. The intensity for white is rather high while the intensity for black is low. By using only the intensity of the image both CPU-cycles and effort can be saved. Mainly since, whether or not the white colour absorb surrounding colours, the intensity for white will still remain high. Thus it will also introduce robustness in this approach. However, there are problems arising from this idea as well. As shown in Figure 6.1, a shadow on a green carpet may have a very low intensity, almost as low as the black colour itself. To avoid this problem, at least to some extent, a histogram combined with thresholding can be used. As it will be discussed later, finding a good threshold for this purpose is crucial.

Figure 6.1: The image formed using only the Y-component.

The algorithm used with the gray-scale image is divided into an image processing part and an image analysis part. The first part describes convolution masks and in particular the Roberts Cross operator. It also explains the use of histograms combined with thresholding in this algorithm. In the second part the concept of blobs is discussed, and also the means for classifying them as a ball or not.

[1]Luminance is not the equivalent of intensity but this thesis makes no difference between them.

6.3.1 Defining and Using the Roberts Cross

One of the first ideas using the gray-scale image was straightforward and simple. The algorithm proposed used only a histogram of the image and then applied two thresholds, I_{low} and I_{high}. These thresholds were meant to filter out all pixels located in the interval created by these values, i.e., only pixels with high/low enough intensity should be saved. The filtering function used can be found in the equation below where I' is the new thresholded image and I is the original gray-scale image.

$$I'_{y,x} = \begin{cases} 0, & I_{y,x} \in [I_{low}, I_{high}]; \\ 1, & I_{y,x} \notin [I_{low}, I_{high}]; \end{cases} , \forall y \in [0, 143], \forall x \in [0, 175]$$

Due to the fact that black seemed to be distributed very well over the image a lot of noise appeared when only using thresholding. The conclusion was that something else had to be added either before or after the thresholding to remove some noise. Instead of using all the gray-scales of the image, a subset consisting of only the transitions between white and black, i.e., strong gradients in the image could be used. In other words, a convolution mask suitable for this problem needed to be discovered.

The Sobel operator was found to be superior in finding the desired features of the ball. However, this operator slowed down the image processing beyond the the real-time demands specified by Team Sweden. Instead of Sobel a smaller and faster operator was tested, namely the Roberts Cross operators R_x and R_y.

$$R_x = \begin{vmatrix} +1 & 0 \\ 0 & -1 \end{vmatrix}, R_y = \begin{vmatrix} 0 & +1 \\ -1 & 0 \end{vmatrix}$$

What the Roberts Cross operators do is calculating the vertical and horizontal gradient. These gradients can be combined using equation (6.7) or, more typically, equation (6.8), which is much faster to compute.

$$G_{xy} = \sqrt{G_x^2 + G_y^2} \tag{6.7}$$

$$G_{xy} = |G_x| + |G_y| \tag{6.8}$$

Applying operators in the manner presented in equation (6.4) is, to say the least, computationally expensive, even with small operators such as this. Instead a Pseudo operator P_{xy} of the Roberts Cross can be derived.

$$P_{xy} = \begin{vmatrix} P1 & P2 \\ P3 & P4 \end{vmatrix}$$

The combined gradient can be calculated by equation (6.9). More specifically the mathematical function provided in equation (6.10) will be used to find all the edges in the gray-scale image.

$$G_{xy} = |P1 - P4| + |P2 - P3| \tag{6.9}$$

$$I'_{y,x} = |I_{y,x} - I_{y+1,x+1}| + |I_{y,x+1} - I_{y+1,x}|, \forall y \in [0, 142], \forall x \in [0, 174] \tag{6.10}$$

Thus, using equation (6.10), instead of using equation (6.4) for R_x and R_y and then combining them by $R_{xy} = |R_x| + |R_y|$, saves a lot of processing time.

6.3.2 Analysing the Histogram

By first applying the P_{xy} operator to the image in Figure 6.1 we get a gray-scaled image featuring only edges. The results from the operator can be found in Figure 6.2. Hopefully the black patches should give us one circle each. The next step is to look at the histogram of the new image and decide a good threshold for it. The reason for the thresholding is because a binary image is always easier to handle and analyse than a gray-scaled one. Since the ball is very small we typically want intensities with few pixels present. But not too few since they are probably nothing more than noise. As seen in Figure 6.3 the histogram has quite a lot of low intensity pixels and we get a slope towards the brighter pixels. Somewhere along this slope is the wanted threshold.

Figure 6.2: The image after applying the Roberts Cross operators.

Since images will undoubtedly differ greatly in intensity values there is no possibility of just deciding upon a fixed threshold value. In fact, consecutive images of the same configuration with the same lighting conditions can have very

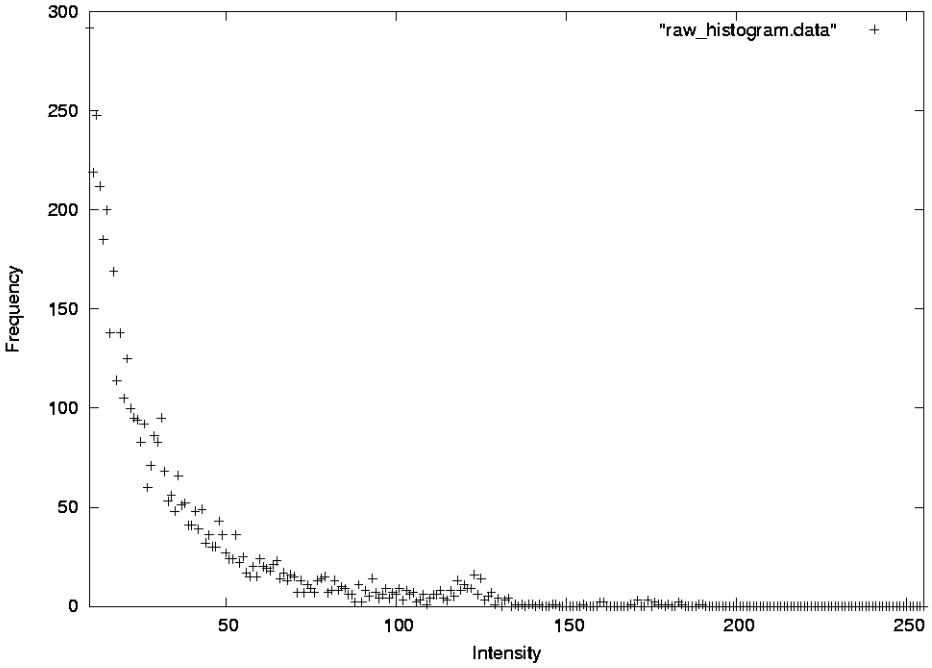

Figure 6.3: A histogram of a typical image of the field and the ball.

different intensity values. Hence there is a need for a thresholding function that can decide which value in the histogram to choose. It must take into account that too high a threshold discards too much information, meanwhile a much lower value may present us with a lot of noise which will be difficult to filter out. There is always a reasonable threshold to apply, and that threshold can be found where the slope on Figure 6.3 flattens, since that is when most of the darker intensities, that are typical noise, are left behind. So basically the function should scan the histogram and measure the slope at every iteration. When the slope is less than or equal to $k_{max} = 0.02$ it sets the threshold at the current intensity. The value of k_{max} was derived by manually setting the best threshold in 100 sample images and then calculating the mean k_{max} value. This seems to be the place where enough noise is lost but still the vital parts of the ball is left intact. The result of the thresholding was still not very good due to the many local slopes in the histogram. In order to get rid of these local slopes, and get the histogram shown in Figure 6.4, the following averaging function is applied:

$$h(x) = \frac{\sum_{n=x}^{x+4} h(n)}{5}, \quad \forall x \in [0, 251]$$

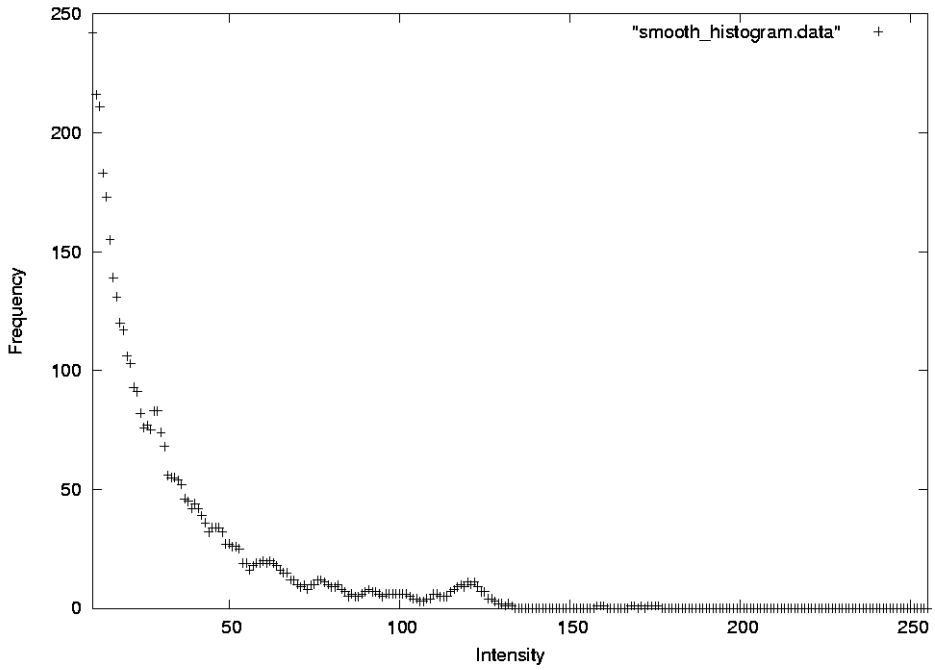

Figure 6.4: A smoothened histogram of a typical image of the field and the ball.

6.3.3 The Creation of Blobs

At this point we should have a binary image, like the one found in Figure 6.5, with at least three white circles originating from the black patches on the ball. The next step is to identify all possible blobs in the image. This is accomplished by using the algorithm defined below.

1. Scan the binary image line by line until an on-pixel[2] is found. If this pixel does not belong to an already classified blob them move to the next step. Otherwise keep scanning.

2. Search down, left and right 11 steps for another on-pixel. If such a pixel is found in all the mentioned directions the algorithm goes into the next step. If not, return to step one.

3. Initiate a flood-fill algorithm [24] on the pixel located between the upper and the lower on-pixel. If the flood-fill does not grow above the maximum allowed blob size, the area filled by the flood-fill algorithm is classified as a blob and put in a blob list. Otherwise go to step one.

[2]The on-pixel belongs to one of the edges detected by the Roberts Cross operator.

Figure 6.5: The image after applying a threshold.

When all of the image has been scanned a blob filtering algorithm is initiated. It searches through the blob space and checks that all blobs are within a certain range of size, i.e., not too big and not too small. By using Gnuplot for plotting all the blobs, and a wide variety of images, the conclusion that blobs should have a radius of at least 1.5 pixels and at most 8 pixels was made.

6.3.4 Recognising the Ball

The fundamental idea is to take advantage of the fact that there are always three or more black patches/blobs visible on the ball from every angle[3]. Also the centre of these three blobs should form an isosceles triangle, which gives us a possibility to further check that the blobs are really part of the ball and not some isolated shadow somewhere. This means that the algorithm has to scan through the blob space creating a list of every possible 3-tuple. When deciding whether a tuple is part of a ball or not we have to calculate the angles in the triangle that is made up from three candidate blobs A, B and C, and check that they are $60° \pm \theta$ where $\theta = 10°$ is the angle of tolerance. The angles are calculated by using the vectors between every endpoint in the triangle. An example of the calculation is given in equation (6.11), where the x and y values are taken from the centre of each blob, i.e., blob A has its centre at (x_1, y_1) etc. If a tuple passes this angle test it is classified as a ball, and the resulting image looks similar to the one shown in Figure 6.6.

$$cos\ \alpha = \frac{u \cdot v}{|u| * |v|} \tag{6.11}$$

[3]This condition does not hold when the ball is very close to the camera.

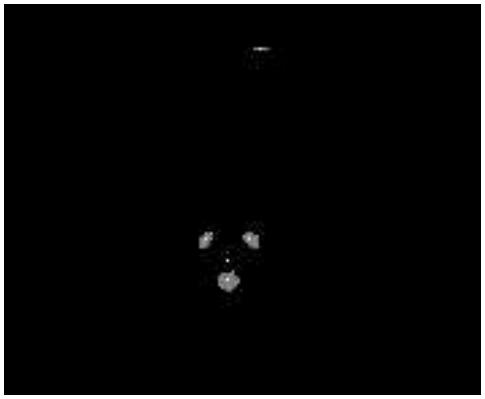

Figure 6.6: The fully analysed image.

$$u = \vec{AB} = B - A = (x_2 - x_1, y_2 - y_1) \tag{6.12}$$

$$v = \vec{AC} = C - A = (x_3 - x_1, y_3 - y_1) \tag{6.13}$$

6.4 Using a Colour Image

This section deals with the ball recognition using all three components in the YCbCr colour space. All of the preprocessing of the image is done by the Pam in the Team Sweden code, which means that this approach only had to investigate an already created blob space, and see what could be concluded from it.

The processing, i.e., the segmentation, part is rather straightforward and simple. It uses the colour segmentation provided by AIBO and OPEN-R. Due to the hardware limitations it can only segment on 8 different colours, so called channels. Also it can only segment on the medium(88x72) image resolution provided by OPEN-R. One major advantage though is that it can run in real time at a frame rate of 25 Hz. This frame rate decreases severely during the region growing that Team Sweden uses in order to get clearer perception of the objects. Due to the fact that all objects are classified by the shape, size, colour and position of their 2D projection, i.e., the blob in the image, it is extremely important to get as many pixels as possible correctly classified. This way the bounding box of the blob might actually be a good representation of the object at hand. Moreover, Team Sweden uses also a blob merging algorithm in order to compensate for poor colour tables, lighting fluctuations, etc., which of course also decreases the frame rate.

During the definition of the problem with recognising the black and white ball the conclusion was made that a ball is represented by white and black blobs. To be more specific, a ball consists of one white blob and one or more enclosed black

blobs. However, since colour tables for white and black are hard to create, it is very difficult to get good blobs from the region growing. This means that it is quite a challenge, if not intractable, to get all of the pixels belonging to the ball correctly classified. In addition, the camera that AIBO uses will not be able to see more than 10, or even fewer, white pixels when the ball is further than two meters away. Needless to say the black blobs will not show at all. In the worst, and probably the most frequent, case the pixels seen at such a distance will be neither white nor black. Instead they will be grayish due to blending. This leaves us with two basic problems:

1. Recognising the ball at close range, i.e., between zero and one meter, using both the white and the black blobs.

2. Recognising the ball at a greater distance using only the white blobs.

As the field is a bit larger than $14m^2$, the robot could move around quite a while before it found the ball using only the close range solution. On the other hand, only using the second approach introduces more problems, e.g., telling the difference between the ball and the border when the ball is really close to the camera so that it occupies 75% or more of the image. Obviously both of these problems must be solved and incorporated in some sort of heuristics.

6.4.1 Long Range Algorithm

When searching for a ball that is far away there are not many features available to depend upon. In fact the only things that can be used is the colour and the shape of the blob. So the question is: How can these attributes be used in order to detect the ball? Well, the colour is obvious since a blob can only consist of one. In other words if a white blob is found it is a potential ball and has to be investigated. However, white blobs will be present in almost every image the camera takes since both the field lines and the borders are white. Thus the colour does not really narrow down the search that much.

Looking at the shape of the white blob will reveal more. In order to use the shape as an attribute, some sort of circle detection algorithm would have to be used, e.g., the Hough Transform. One of the major problems when dealing with AIBO is the low computational power which means that expensive algorithms, such as the Hough Transform hardly can be used. A trigonometric approach for identifying circles would also be computationally very demanding. Instead of determining the actual shape of the blob the bounding box of the blob might do just fine. The fact that a circle will always create a quadratic bounding box can be used to filter out many of the white blobs originating from the field lines or the borders.

Sadly the segmentation rarely classifies all of the white pixels belonging to the ball correctly. This happens due to many factors but primarily because the carpet in the field makes the ball look green in the lower part. Thus the correspondence between the width and the height of the white blob is $1 : x$ rather than $1 : 1$. This means that the algorithm should not be looking for a quadratic white blob but a rectangular one having $height = x * width$. By plotting a considerable number of white blobs originating from the ball the value for x could be experimentally determined to 0.8. Having this value actually completes this method. However x was determined through mean value which implies that more rectangles than the ones having exactly $height = x * width$ should be classified as a ball. Thus a match function is needed to determine how well the bounding box at hand matches the desired rectangle condition. This presented a new value to tweak. How well should the rectangles match? It turned out that a match of 80% was enough to get many of the white blobs correctly classified. This ratio was decided by looking at the number of false positives divided by the true positives, and a match condition of 80% simply gave the lowest quotient, i.e., about 1-2 true positives per false positive.

There is a need for one more condition still, namely the size. If the ball is supposed to be far away the blob can hardly take up a large part of the image. Thus, the last condition required here is that the blob is small enough. There will not be a specified value for this condition since it was extremely dependant on the lighting. In other words no good and general value could be found. Instead it was tweaked specifically on site for Lund, Örebro and Padova.

To summarise, the following is done, for every blob in the blob space, in order to detect a distant ball:

1. Check that the blob is white.
2. Check that there is at least an 80% match to the rectangle condition.
3. Check that the blob is small enough.

This is about all that can be done in the long range ball detection, since there are so few features available. Using this approach the robot could recognise the ball about 50% of the time.

6.4.2 Close Range Algorithm

When the ball is close, or at least not too far away, there are a number of features available for classification. The ones used in this approach are:

1. Colour and size of the blob.

2. Quadratic ratio for the blob.

3. Quantity and colour of contained blobs.

4. Size ratio between the blob and the contained ones.

The first two items in the list were explained in section 6.4.1 and will be left out in this description. The next feature is rather obvious since a ball located close to the camera can have one to five visible black blobs, no more and no less. What size ratio is feasible to use for the black and white blobs? To answer that question, the diameter of the entire ball d_{ball} along with the diameter of one of the black patches d_{patch}, is needed. The ratio $d_{ratio} = d_{patch}/d_{ball}$ was found to be 0.25. In the segmented images, however, the ratio between the blobs were closer to 0.20 due to the already mentioned problems with white and black colours. Just like the quadratic condition it is advisable to allow fluctuations in the size ratio as well. Hence a match of 80% is enough to pass the size ratio condition which means that ratios between 0.16 and 0.24 are accepted.

With all this information at hand, it might appear as if locating the ball was an easy task. To answer it negatively, a plot is presented in Figure 6.7. The plot describes the fluctuations in the match value of the quadratic ratio and the black/white ratio. All of these 46 consecutive images are captured during the exactly same conditions, which means that the ball and the robot were fixed in one position and in that position all 46 images were taken. The lighting conditions were also the same. In other words this plot describes the performance of the camera located in AIBO. The results are rather intimidating.

6.5 Results and Implementation

Examining the plot in Figure 6.7 reveals that the quadratic match fluctuates between 63% and 85%. It also seems to be somewhat periodic. This effect creates interlacing on the images captured by AIBO and it originates from the frequency of the spotlights at RoboCup 2003. Sometimes the interlacing effect was substantial enough to produce black horizontal lines through the image, which in many cases passed through the ball. Naturally if such a line divides the white blob, originating from the ball, it will create two white blobs. Each of them not passing the quadratic condition. This condition is present in both the long range approach and the close range approach. Thus both of them suffer in performance from this interlacing.

The phenomenon with the black and white match, can be explained by this effect as well. There is a difference though, the black and white match is generally lower than the quadratic. The reason is simple, a black line crossing the whole

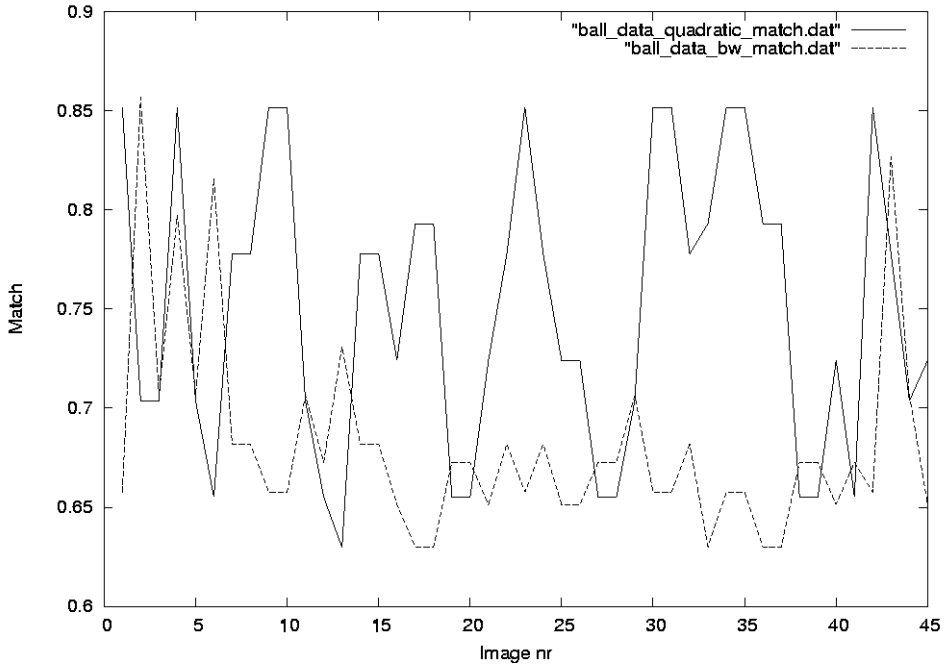

Figure 6.7: Fluctuations in the match ratio of the black/white condition and the quadratic condition.

image, through the ball, will yield a big black blob. That black blob will be often be larger, using area, than 25% of the white blob thus creating a poor match with the black and white ratio condition. A 3D plot, shown in Figure 6.8, of the same data series as the previous plot reveals the disperse nature of the matchings.

The implementation as such was prototyped in Python [5] and later ported to C++. The code for the gray-scale approach is listed in Appendix C, and the implementation using a colour image is listed in Appendix D. It should be mentioned that this code has been working in cooperation with the Team Sweden code and cannot be used as is. At least not without some major modifications.

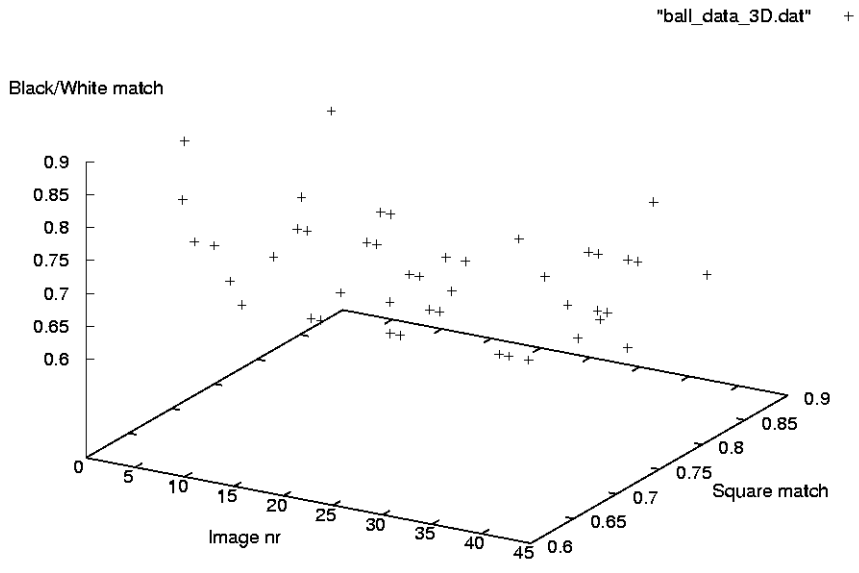

Figure 6.8: Fluctuations in the match ratio of the black/white condition and the quadratic condition.

Chapter 7

Conclusions and Future Improvements

In this thesis two approaches for solving the same problem have been presented. Each of them has its own strengths and weaknesses. This section will try to sort out the pros and cons for the gray-scale versus the colour approach.

7.1 Gray-scale Approach

This was an attempt to throw away quite a lot of information from the beginning, in order to get results fast. It failed in a very important aspect though. The algorithm developed is not capable of recognising the ball at all distances since it relies on the assumption that at least three black patches are always visible. As previously discussed, this is often not the case. In fact, this algorithm would only recognise the ball at a semi-close distance which means that the robot would loose track of the ball as soon as it came close enough. In other words, touching the ball, let alone kicking it into a net, would be an act of pure luck using this approach. Looking at the bright side, the algorithm is quite fast despite the use of convolution masks. It is also rather insensitive to lighting fluctuations since it triggers on transitions between high and low intensities in the image. Also the tiresome work of creating colour tables is not necessary for obvious reasons.

7.2 Colour Approach

Using the coloured image presented a lot more features to use when searching for, and identifying the ball. All of these features in turn provided ball recognition abilities at any distance up to three meters which means that the robot is actually

capable of deliberately touching and kicking the ball. There are a few downsides to this solution though. First it demands really good colour tables, i.e., 60 pictures or more of every object in the field need to be manually tuned using a rather crude tool. Second, even with very good colour tables this approach is still highly sensitive to different lighting conditions. For example, if one out of four spotlights goes out, a whole new colour table would need to be created. In fact, these tables had to be reproduced for every new site that the robot is supposed to function in. In other words, there are several colour tables for Lund, Ronneby, Murcia and Örebro.

7.3 Challenge 1

Because of the previously discussed limitations of the gray-scale approach, our solution for Challenge 1 used the colour approach. Due to an administrative misunderstanding a version which did not use the long distance ball detection was used. Hence it took a while for the robot to detect the ball and touch it. After detecting the ball only the time for one attempt to kick the ball was left. The kick was a complete failure due to bad distance calibration and also because the ball was actually at the border which makes the white segmentation useless. After this attempt to kick the ball the three minutes available for the challenge were over. This was a rather difficult challenge judging from the overall, results since no team managed to score and $2/3$ of the teams did not even touch the ball. The efforts presented in this thesis led to a 7:th place out of 26 competitors.

7.4 Suggested Improvements

When it comes to the proposed way of solving the problem with the gray-scale image, there are in my opinion very few things to improve since the basic assumption, that there are always three or more black patches visible, is fundamentally wrong. Maybe it could be used as a separate part in a more extensive algorithm. However, it can not alone be made to detect the ball at an arbitrary distance.

The colour approach can be improved in many ways, e.g., by changing the colour representation from YCbCr to another, better separated colour space [15], or using scan-lines [16]. Also the colour tables could be tweaked “on-line” instead of manually tweaking images on a nearby computer. Moreover the algorithm performs rather poor when the ball is close to the border since it uses a rough way of estimating the actual ball size by looking at the size and position of the black blobs at hand. One way of improving this might be to take the distance between these black blobs into account, thus improving the ball detection rate.

Bibliography

[1] Web site of RoboCup:
http://www.robocup.org
(Verified 15th August 2003)

[2] Web site of Sony Legged Robot League:
http://www.openr.org/robocup/index.html
(Verified 15th August 2003)

[3] Web site of AIBO:
http://http://www.aibo-europe.com
(Verified 15th August 2003)

[4] Web site of the Thinking Cap architecture:
http://www.aass.oru.se/ asaffio/Software/TC/
(Verified 15th August 2003)

[5] Web site of the Python programming language:
http://www.python.org
(Verified 15th August 2003)

[6] OPEN-R SDK: Model Information for ERS-210, Sony Computer Science Laboratory Inc., 2002.

[7] A. Saffiotti, A. Björklund, S. Johansson, Z. Wasik: Team Sweden, A. Birk, S. Coradeschi, S. Tadokoro (Eds) RoboCup 2001, Springer Verlag, 2002.

[8] Z. Wasik, A. Saffiotti: Robust Colour Segmentation for the RoboCup Domain, Pattern Recognition, Proc. of the Int. Conf. on Pattern Recognition (ICPR), volume 2, pages 651-654, 2002.

[9] A. Saffiotti, K. LeBlanc: Active Perceptual Anchoring of Robot Behaviour in a Dynamic Environment, Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA), volume 4, pages 3796-3802, 2000.

[10] A. Saffiotti, Z. Wasik: Using Hierarchical Fuzzy Behaviours in the RoboCup Domain, C. Zhou, D. Maravall and D. Ruan (EDS), Autonomous Robotic Systems, pages 235-262, Springer, DE, 2003.

[11] S. Johansson, A. Saffiotti: Using the Electric Field Approach in the RoboCup Domain, A. Birk, S. Coradeschi, S. Tadokoro (EDS), Robot Soccer World Cup V, pages 399-404, Springer-Verlag, DE, 2002.

[12] B. Hengst, B. Ibbotson, P. Pham, C. Sammut: Omnidirection - a Locomotion for Quadruped Robots, A. Birk, S. Coradeschi, S. Tadokoro (EDS), RoboCup 2001, Springer Verlag, 2002.

[13] R. Adams, L. Bischof: Seeded Region Growing, Pattern Analysis and Machine Intelligence, IEEE Transactions on, volume 16, issue 6, pages 641-647, 1994.

[14] Yasuhiko Yokote: The Apertos Reflective Operating System: The Concept and Its Implementation, Sony Computer Science Laboratory Inc., 1992.

[15] Gerd Mayer, Hans Utz, Gerhard K. Kraetzschmar: Playing Robot Soccer under Natural Light: A Case Study, RoboCup Symposium, 2003.

[16] M. Jüngel, J. Hoffmann, M. Lötzsch: A Real-Time Auto-Adjusting Vision System for Robotic Soccer, Pre-proceedings of RoboCup Symposium, 2003.

[17] K. K. Pingle: Visual perception by computer, In A. Grasselli, editor, Automatic Interpretation and Classification of Images, pages 277-284, Academic Press, New York, 1969.

[18] L. G. Roberts: Machine Perception of Three-dimensional Solids, In J. T. Tippet et al., editor, Optical and Electro-Optical Information Processing, pages 159-197, MIT Press, Cambridge, 1965.

[19] J Canny: A Computational Approach to Edge Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 8:679-698, 1986.

[20] R. Gonzalez and R. Woods Digital Image Processing, Ch 7, Addison-Wesley Publishing Company, 1992.

[21] R. R. Murphy: Introduction to AI Robotics, pages 220-231, MIT Press, 2000.

[22] J. Bruce, T. Balch, M. Veloso: Fast and Cheap Colour Image Segmentation for Interactive Robots, Intelligent Robots and Systems (IROS 2000), 2000.

[23] S. Russel, P. Norvig: Artificial Intelligence, Ch 2.4, Prentice-Hall, 1995.

[24] A. Glassner: Fill 'er up! [Graphics filling algorithms], Computer Graphics and Applications, IEEE, Volume: 21 Issue: 1, pages 78-85, 2001.

Appendix A

Terminology

Autonomous agent: A hardware/software agent whose behaviour is determined by its own experience.

Histogram: A plot describing frequency of pixels within every intensity of the gray-scale image.

YCbCr: A colour space model also known as YUV where Y is the luminance, U(Cb) is the shift from red and V(Cr) is the shift from blue.

A.1 Abbreviations

RISC: Reduced Instruction Set Computer; one whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions.

MIPS: A microprocessor vendor.

SDK: Software Development Kit.

PCM: Pulse Code Modulation; a general digital audio processing with volume samples generated in continuous time periods.

CMOS: Complementary Metal Oxide Semiconductor; a type of memory that runs on very little power.

FAT-16: File Allocation Table; a file-system created by Microsoft using 16 bit disk addresses.

Appendix B

Installation and Configuration of OPEN-R SDK

B.1 Background

The reason for this project was mainly to replace the Windows 2000 and Cygwin environment with Red Hat Linux 8 as a platform when developing software using the OPEN-R SDK[1]. According to former thesis writers the Windows/Cygwin environment has been, to say the least, frustrating to work with. Some of them even compiled their code in Linux and later started Windows 2000 in order to transfer their compiled objects to the memory stick[2]. The goal in short terms was to be able to fully utilise the OPEN-R SDK from Linux. Please note that this document is in no way trying to create a substitute for the installation instructions from the official OPEN-R SDK documentation.

B.2 Solution

Several steps were required to fulfil the goal defined above and each of these steps will be described in detail. In order to retrieve the files needed to install the OPEN-R SDK a registration form located at its web page[3] must be filled out. From the download section on the page the files listed in table B.1 will be needed. Please note that the version numbers may of course change as this document grows old.

[1] Sony's OPEN-R Software Development Kit
[2] We are referring to the AIBO Programming Memory Stick
[3] www.aibo.com/openr

binutils-2.13.tar.gz Binary utilities for developing software.

newlib-1.10.0.tar.gz A stripped libc really.

gcc-3.2.tar.gz The compiler needed for obvious purposes.

build-devtools-3.2-r1.sh Installation script.

OPEN_R_SDK-1.1.3-r2.tar.gz Standard configuration for memory stick.

OPEN_R_SDK-sample-1.1.3-r3.tar.gz Sample programs.

Table B.1: List of files needed for installation.

B.2.1 Installing the Development Tools

Root access is demanded in order to install the development tools since the install script needs to access the local machine's `/usr/local` directory. The software can be installed in other directories though but it is not recommended. Hence root access is assumed throughout this document.

For this installation part the top four files listed in table B.1 are needed. It does not matter where these files are stored, i.e., a home directory will do just fine. By executing the steps described below the development tools are installed.

- Change directory to where the four files mentioned above are stored.
- Type `su` and press enter and provide the root password and press enter again.
- Execute the script that builds the development tools by typing: `./build-devtools-3.2-r1.sh` and press enter.

Within a few minutes the basic development tools should have been installed.

B.2.2 Installing the OPEN-R SDK

This section describes the installation procedure of the basic configuration possibilities for the memory stick. These should be unpacked in the `/usr/local` directory. It contains the real-time OS Aperios made specifically for AIBO by Sony and some default libraries used by OPEN-R. Once again the steps needed are described below. described below. Note that root access is still needed.

- Type `cd /usr/local`.

- Type `tar -zxvf /exampledir/OPEN_R_SDK-1.1.3-r2.tar.gz` where exampledir is the directory where the files listed in Table B.1 are stored.

Now this part of the installation is done and there is no need for further root access so type `exit` and press enter.

B.2.3 Installing the Sample Programs

This installation is merely an unpacking of a compressed directory. Type `tar -zxvf OPEN_R_SDK-sample-1.1.3-r3.tar.gz` in the directory containing the downloaded files from table B.1. A directory sample will be created and in that directory many sample programs can be found.

B.2.4 Mounting the MSAC-US1 Device

This consists basically of one command but some effort is put into understanding the command as well as using it in this section. The `fstab` file along with the `mount` command will be used, i.e., a `man mount` command is a good place to begin. There were a few problems arising at the first mount attempt and the problem was that the mount command could not auto detect the file-system on the memory stick. After a few minutes it was discovered that the windows file system FAT was the correct one. In order to mount any device under Linux there are two major questions.

1. What entry under the `/dev` catalogue is the device connected to?
2. Where shall the device be mounted?

The answer to the first question is `/dev/sda1` due to the fact that the Linux kernel emulates many USB devices as a SCSI device. The sda1 stands for special device 1. The next question is a matter of preference but in this report we use the `/mnt/msac` directory to serve as a mounting point for the memory stick. At this point the device can be mounted by a super user by executing the following command:

```
mount -t vfat -rw /dev/sda1 /mnt/msac
```

In order to let users mount this device a line in `fstab`[4] needs to be added. The line looks like this:

[4] fstab tells the kernel which devices and file systems that can be mounted and by who.

/dev/sda1 /mnt/msac vfat noauto,user,rw 0 0

The details of this will not be explain in this document. For the interested there is a man page for mount to read. This means that any user on Clarke.ludat.lth.se can type `mount /mnt/msac`. The `/mnt/msac` directory then serves as the root directory for the memory stick, which means that files can be removed from and copied to it.

B.3 Results

Everything is working in the current Linux distribution installed on Clarke.ludat.lth.se. The distribution at hand here is Red Hat 8.0. It has been my experience that this is not one of Red Hat's more successful releases but there were no major problems encountered during the installation and configuration of the OPEN-R SDK. In other words compilation and transferring of source code to a memory stick using the MSAC-US1 from Sony can be done on Clarke.

Appendix C

C++ Code for the Gray-scale Approach

RecognizeBall.cc

```
#include <stdio.h>
#include <math.h>
#include "Table.h"
#include "Constants.h"
#include "RecognizeBall.h"

/////// PRIVATE ///////

/* UTILITIES */

    int
RecognizeBall::Fac(int n)
{
    int result = 1;
    while(n > 0){
        result = result * n;
        n = n - 1;
    }
    return result;
}

    int
RecognizeBall::Combination(int n, int k)
```

```
{
    int delta = k - 1;
    int prod = 1;
    while(delta >= 0){
        prod = prod * (n-delta);
        delta--;
    }
    return prod / Fac(k);
    //return Fac(n)/(Fac(k)*Fac(n-k));
}

    void
RecognizeBall::KillBloblist(bloblist_t *bloblist)
{
    int cntr = 0;
    while(bloblist != NULL){
        bloblist_t *tmp = bloblist->next;
        if(bloblist->blob != NULL){
            delete bloblist->blob;
        }
        delete bloblist;
        bloblist = tmp;
        cntr++;
    }
}

    int
RecognizeBall::IsElement(int element, int array[], int *arraylen)
{
    int len = *arraylen;
    for(int i=0; i<len; i++)
        if(array[i] == element)
            return 1;
    return 0;
}

    int
RecognizeBall::CountBlobs(bloblist_t *bloblist)
{
    bloblist_t *tmp = bloblist;
    int cntr = 0;
```

```
    while(tmp != NULL){
        cntr++;
        tmp = tmp->next;
    }
    return cntr;
}


/* ANALYSIS SECTION */

    int
RecognizeBall::CheckPixel(int x, int y)
{
    return !(x<0 || y<0 || x>=SOBEL_IMAGE_WIDTH ||
            y>=SOBEL_IMAGE_HEIGHT ||
            sImage[YX_TO_INDEX[y][x]]==ON_PIXEL ||
            sImage[YX_TO_INDEX[y][x]]==BLOB_PIXEL);
}

    AdHocList*
RecognizeBall::MakeBlob(int offset)
{
    AdHocList *blob = new AdHocList();
    AdHocList *head = new AdHocList();
    const int *coord = INDEX_TO_YX[offset];

    if(CheckPixel(coord[X_INDEX],coord[Y_INDEX])){
        head->Push(offset);
        blob->Push(offset);
        sImage[offset] = BLOB_PIXEL;
    }

    while(head->Size() > 0){
        offset = head->Pop();

        if(blob->Size() > MAX_BLOB_SIZE){ //Rollback action
            while(blob->Size() > 0)
                sImage[blob->Pop()] = OFF_PIXEL;
            while(head->Size() > 0)
                sImage[head->Pop()] = OFF_PIXEL;
            delete head;
```

```
            delete blob;
            return NULL;
        }
        coord = INDEX_TO_YX[offset];
        int y = coord[Y_INDEX];
        int x = coord[X_INDEX];
        int index = -1;

        if(CheckPixel(x-1,y)){
            index = offset - 1;
            head->Push(index);
            sImage[index] = BLOB_PIXEL;
            blob->Push(index);
        }
        if(CheckPixel(x,y-1)){
            index = offset - SOBEL_IMAGE_WIDTH;
            head->Push(index);
            sImage[index] = BLOB_PIXEL;
            blob->Push(index);
        }
        if(CheckPixel(x+1,y)){
            index = offset + 1;
            head->Push(index);
            sImage[index] = BLOB_PIXEL;
            blob->Push(index);
        }
        if(CheckPixel(x,y+1)){
            index = offset + SOBEL_IMAGE_WIDTH;
            head->Push(index);
            sImage[index] = BLOB_PIXEL;
            blob->Push(index);
        }
    }
    delete head;
    if(blob->Size() < MIN_BLOB_SIZE){
        while(blob->Size() > 0)
            sImage[blob->Pop()] = OFF_PIXEL;
        delete blob;
        blob = NULL;
    }
    return blob;
```

```
}


    int
RecognizeBall::CheckPerimeter(int offset)
{
    int left = 0;
    int right = 0;
    const int *coord = INDEX_TO_YX[offset];
    int y = coord[Y_INDEX];
    int x = coord[X_INDEX];
    for(int i=0; i<MAX_DISTANCE_TO_PIXEL; i++){
        if(x+i < SOBEL_IMAGE_WIDTH && sImage[offset+i] == ON_PIXEL)
            right = 1;
        if(x-i >= 0 && sImage[offset-i] == ON_PIXEL)
            left = 1;
    }
    return right && left;
}

    int
RecognizeBall::FindPotentialBlob(int offset)
{
    int blackStart = -1;
    int blackStop = -1;
    int half = -1;
    for(int i=1; i<MAX_DISTANCE_TO_PIXEL; i++){
        int index = offset+i*SOBEL_IMAGE_WIDTH;
        if(index >= SOBEL_IMAGE_SIZE)
            return -1;
        byte pixel = sImage[index];
        if(blackStart < 0 && pixel == OFF_PIXEL){
            blackStart = index;
        }
        else if(blackStart >= 0 && blackStop < 0 && pixel == ON_PIXEL){
            blackStop = index - SOBEL_IMAGE_WIDTH;
            half = blackStart +
                (((blackStop - blackStart) / 175) / 2) * 175;
            if(CheckPerimeter(half)){
                return half;
            }
```

```
        }
    }
    return -1;
}

    int
RecognizeBall::FindNextOnPixel(int offset)
{
    for(int i=offset; i<SOBEL_IMAGE_SIZE; i++)
        if(sImage[i] == ON_PIXEL)
            return i;
    return -1; //no more on pixels in the image
}

    bloblist_t*
RecognizeBall::FindAllBlobs()
{
    bloblist_t *bloblist = new bloblist_t(NULL, NULL);
    bloblist_t *currblob = bloblist;
    int onPixel = FindNextOnPixel(0);
    while(onPixel >= 0){
        int regionGrowingOffset = FindPotentialBlob(onPixel);
        if(regionGrowingOffset > -1){
            AdHocList *tmpBlob = MakeBlob(regionGrowingOffset);
            if(tmpBlob != NULL){
                currblob->next =
                    new bloblist_t(tmpBlob, NULL);
                currblob = currblob->next;
            }
        }
        onPixel = FindNextOnPixel(onPixel+1);
    }
    currblob = bloblist->next;
    delete bloblist;

    return currblob;
}

    AdHocList*
RecognizeBall::CalcBlobCenter(bloblist_t *bloblist)
{
```

```
    bloblist_t *curr = bloblist;
    AdHocList *blobcenter = new AdHocList();
    const int *yx;

    while(curr != NULL){
        AdHocList *blob = curr->blob;
        yx = INDEX_TO_YX[blob->Max()];
        int yMin = yx[Y_INDEX];
        yx = INDEX_TO_YX[blob->Min()];
        int yMax = yx[Y_INDEX];
        int ycenter = yMin + (yMax - yMin)/2;
        int xMin = 1000;
        int xMax = -1000;
        for(int i=0; i<blob->Size(); i++){
            yx = INDEX_TO_YX[blob->Get(i)];
            int x = yx[X_INDEX];

            if(x > xMax)
                xMax = x;
            if(x < xMin)
                xMin = x;
        }
        int xcenter = xMin + (xMax - xMin)/2;
        blobcenter->Push(YX_TO_INDEX[ycenter][xcenter]);
        curr = curr->next;
    }
    return blobcenter;
}

    float
RecognizeBall::CalcDistanceToBall(int dist)
{
    return dist * SCALE_TO_REAL_WORLD;
}

    point_t
RecognizeBall::CalcCenterOfBall(point_t p1, point_t p2, point_t p3)
{
    int Minx = MIN(p1.x, p2.x);
    Minx = (Minx < p3.x) ? Minx : p3.x;
    int Maxx = MAX(p1.x, p2.x);
```

```
    Maxx = (Maxx > p3.x) ? Maxx : p3.x;
    int Miny = MIN(p1.y, p2.y);
    Miny = (Miny < p3.y) ? Miny : p3.y;
    int Maxy = MAX(p1.y, p2.y);
    Maxy = (Maxy > p3.y) ? Maxy : p3.y;
    int cx = Minx + (Maxx-Minx)/2;
    int cy = Miny + (Maxy-Miny)/2;
    point_t p;
    p.x=cx;
    p.y=cy;

    return p;
}

    float
RecognizeBall::CalcAngle(vector_t u, vector_t v)
{
    float dotprod = u.x * v.x + u.y * v.y;
    float ulen = sqrt((double)(u.x*u.x + u.y*u.y));
    float vlen = sqrt((double)(v.x*v.x + v.y*v.y));
    return dotprod/(ulen*vlen);
}

    vector_t
RecognizeBall::CalcVector(point_t p1, point_t p2)
{
    vector_t vector(p2.x-p1.x, p2.y-p1.y);
    return vector;
}

    int
RecognizeBall::CheckAngles(point_t p1, point_t p2, point_t p3)
{
    vector_t u = CalcVector(p1,p2);
    vector_t v = CalcVector(p1,p3);
    float angle1 = CalcAngle(u,v);
    u = CalcVector(p2,p1);
    v = CalcVector(p2,p3);
    float angle2 = CalcAngle(u,v);
    u = CalcVector(p3,p1);
    v = CalcVector(p3,p2);
```

```
    float angle3 = CalcAngle(u,v);

    //# The reason for the inverted comparisment is
    //# Due to the fact that cos(70) < cos(50)
    if(angle1 < MAX_ANGLE || angle1 > MIN_ANGLE)
        return 0;
    if(angle2 < MAX_ANGLE || angle2 > MIN_ANGLE)
        return 0;
    if(angle3 < MAX_ANGLE || angle3 > MIN_ANGLE)
        return 0;
    return 1;
}

    float
RecognizeBall::CalcDistance(point_t p1, point_t p2)
{
    return (float)sqrt(pow((double)(p1.x-p2.x),(double)2) +
            pow((double)(p1.y-p2.y),(double)2));
}

    int
RecognizeBall::CheckDistances(point_t p1, point_t p2, point_t p3, float *avg)
{
    float dist1 = CalcDistance(p1,p2);
    float dist2 = CalcDistance(p1,p3);
    float dist3 = CalcDistance(p2,p3);
    *avg = (dist1+dist2+dist3) / 3.0;
    if(dist1 > *avg + DISTANCE_TOLERANCE || dist1 < *avg - DISTANCE_TOLERANCE)
        return 0;
    if(dist2 > *avg + DISTANCE_TOLERANCE || dist2 < *avg - DISTANCE_TOLERANCE)
        return 0;
    if(dist3 > *avg + DISTANCE_TOLERANCE || dist3 < *avg - DISTANCE_TOLERANCE)
        return 0;
    return 1;
}

    void
RecognizeBall::MakeTriangles(int array[], int arraylen, triangle_t result[])
{
    int nbrOfCombs = Combination(arraylen,3);
    int i = 0;
```

```
    int j = 1;
    int k = 2;
    for(int cntr=0; cntr<nbrOfCombs; cntr++){
        const int *yx1 = INDEX_TO_YX[array[i]];
        const int *yx2 = INDEX_TO_YX[array[j]];
        const int *yx3 = INDEX_TO_YX[array[k]];
        point_t p1(yx1[X_INDEX], yx1[Y_INDEX]);
        point_t p2(yx2[X_INDEX], yx2[Y_INDEX]);
        point_t p3(yx3[X_INDEX], yx3[Y_INDEX]);
        triangle_t triangle(p1, p2, p3);
        result[cntr] = triangle;
        k += 1;
        if(k >= arraylen){
            j += 1;
            k = j + 1;
            if(j + 1 >= arraylen){
                i += 1;
                j = i + 1;
                k = j + 1;
                if(i + 2 >= arraylen){
                    return;
                }
            }
        }
    }
}

    int
RecognizeBall::AnalyzeImage(AdHocList *blobcenter, ball_t *ball)
{
    int numblobs = blobcenter->Size();
    int *blobarray = new int[numblobs];
    blobcenter->ToArray(blobarray);
    int numcombs = Combination(numblobs,3);
    triangle_t *triangles = new triangle_t[numcombs];
    MakeTriangles(blobarray,numblobs,triangles);
    delete [] blobarray;

    for(int i=0; i<numcombs; i++){
        triangle_t triangle = triangles[i];
        float avg = 0;
```

```
        if(CheckDistances(triangle.p1, triangle.p2, triangle.p3, &avg)){
            if(CheckAngles(triangle.p1, triangle.p2, triangle.p3)){
                point_t p = CalcCenterOfBall(triangle.p1,
                        triangle.p2,triangle.p3);
                ball->x = p.x;
                ball->y = p.y;
                ball->size = avg;// * SCALE_TO_REAL_WORLD;
                delete triangles;
                return 1;
            }
        }

    }
    delete triangles;
    return 0;
}


/* IMAGE PROCESSING */

    byte*
RecognizeBall::ApplyThreshold(int threshold)
{
    for(int i=0; i<SOBEL_IMAGE_SIZE; i++)
        if(sImage[i] > threshold)
            sImage[i] = ON_PIXEL; //white
        else
            sImage[i] = OFF_PIXEL; //black
    return sImage;
}

    int
RecognizeBall::CalculateThreshold()
{
    int MaxVal = 0;
    int MaxInd = 0;
    for(int i=0; i<NUMBER_OF_GREYS; i++)
        if(MaxVal < histogram[i]){
            MaxVal = histogram[i];
            MaxInd = i;
        }
```

```
    int startPoint = MaxInd;
    int endPoint = MaxInd + INTERVAL;
    while(endPoint < NUMBER_OF_GREYS){
        int deltaX = INTERVAL; //endPoint - startPoint
        int deltaY = histogram[endPoint] - histogram[startPoint];
        float k = 0;
        if(deltaX == 0)
            k=deltaY;
        else
            k = (float)deltaY / (float)deltaX; //the slope
        if(k >= K_MIN)
            return startPoint;
        else{
            startPoint += INTERVAL;
            endPoint += INTERVAL;
        }
    }
    return -1; //We are not happy here
}

    int*
RecognizeBall::SmoothenHistogram()
{
    for(int i=0; i<NUMBER_OF_GREYS - SMOOTH_INTERVAL; i++)
        for(int j=0; j<SMOOTH_INTERVAL-1; j++){
            histogram[i]+=histogram[i+j+1];
            //This division is not really needed since
            //the slope will be the same anyway
            histogram[i] = histogram[i] / SMOOTH_INTERVAL;
        }
    return histogram;
}

    int*
RecognizeBall::MakeHistogram()
{
    for(int i=0; i<SOBEL_IMAGE_SIZE; i++)
        histogram[sImage[i]]++;
    return histogram;
}
```

```
    void
RecognizeBall::ClearHistogram()
{
    for(int i=0; i<NUMBER_OF_GREYS; i++)
        histogram[i] = 0;
}

    byte*
RecognizeBall::ApplySobelMask()
{
    int cntr = 0;
    int offset = 0;
    for(int i=0; i<SOBEL_IMAGE_SIZE; i++){
        if(cntr == ORIG_IMAGE_WIDTH-1){
            offset += 1;
            cntr = 0;
        }
        int row1 = i + offset;
        int row2 = i + ORIG_IMAGE_WIDTH + offset;
        double vertical = yImage[row1] - yImage[row2+1];
        double horizontal = yImage[row1+1] - yImage[row2];
        int result = (int)(sqrt(pow(vertical,(double)2) +
                    pow(horizontal,(double)2)));
        sImage[i] = (byte)result;
        cntr += 1;
    }
}

    void
RecognizeBall::ProcessImage()
{
    ApplySobelMask();
    ClearHistogram();
    MakeHistogram();
    SmoothenHistogram();
    int threshold = CalculateThreshold();
    ApplyThreshold(threshold);
}


/////// PUBLIC ///////
```

```
    int*
RecognizeBall::DetectBall()
{
    ProcessImage();
    bloblist_t *bloblist = FindAllBlobs();
    int nbrOfBlobs = CountBlobs(bloblist);
    if(nbrOfBlobs < 3){
        printf("Found %d blobs ==> Could not detect ball!!\n",
                nbrOfBlobs);
        KillBloblist(bloblist);
        return NULL;
    }
    AdHocList *blobcenterlist = CalcBlobCenter(bloblist);
    FilterWhitePixels(); //DEBUG
    MarkBlobCenter(blobcenterlist);// DEBUG
    ball_t *ball = new ball_t(-1,-1,-1);
    if(AnalyzeImage(blobcenterlist, ball)){
        printf("detected ball at x=%d, y=%d with a size of %f\n"
                ,ball->x, ball->y, ball->size);
        sImage[YX_TO_INDEX[ball->y][ball->x]] = ON_PIXEL;
    }else{
        printf("Could not detect ball!!\n");
    }
    delete ball;
    delete blobcenterlist;
    KillBloblist(bloblist);
    return NULL;
}

    byte*
RecognizeBall::PaddedSobelImage(byte psImage[])
{
    for(int i=0; i<SOBEL_IMAGE_HEIGHT; i++){
        for(int j=0; j<SOBEL_IMAGE_WIDTH; j++){
            int index = i*SOBEL_IMAGE_WIDTH+j;
            psImage[index+i] = sImage[index];
        }
        psImage[i*ORIG_IMAGE_WIDTH+SOBEL_IMAGE_WIDTH] = OFF_PIXEL;
    }
    for(int i=0; i<ORIG_IMAGE_WIDTH; i++)
```

```
        psImage[(ORIG_IMAGE_HEIGHT - 1) * ORIG_IMAGE_WIDTH + i] = OFF_PIXEL;
    return psImage;
}

    byte*
RecognizeBall::SobelImage()
{ return this->sImage; }

    byte*
RecognizeBall::YImage()
{ return this->yImage; }

    void
RecognizeBall::YImage(byte *yImage)
{ this->yImage = yImage; }

RecognizeBall::RecognizeBall(byte *yImage)
{
    this->yImage = yImage;
    this->sImage = new byte[SOBEL_IMAGE_SIZE];
    this->histogram = new int[256];
}

RecognizeBall::~RecognizeBall()
{
    delete [] histogram;
    delete [] sImage;
}
```

RecognizeBall.h

```
#ifndef RecognizeBall_H
#define RecognizeBall_H

#include "AdHocList.h"

#define byte unsigned char
```

```
//Structs
struct bloblist_t
{
    AdHocList *blob;
    bloblist_t *next;

    bloblist_t()
    {
        blob = NULL;
        next = NULL;
    }
    bloblist_t(AdHocList *bl, bloblist_t *ne)
    {
        blob = bl;
        next = ne;
    }
};

struct vector_t
{
    int x;
    int y;

    vector_t(int newx, int newy)
    {
        x = newx;
        y = newy;
    }

};

struct point_t
{
    int x;
    int y;

    point_t(){}
    point_t(int newx, int newy)
    {
        x = newx;
        y = newy;
```

```
    }
};

struct triangle_t
{
    point_t p1;
    point_t p2;
    point_t p3;

    triangle_t(){}
    triangle_t(point_t newp1, point_t newp2, point_t newp3)
    {
        p1 = newp1;
        p2 = newp2;
        p3 = newp3;
    }
};

struct ball_t
{
    int x;
    int y;
    float size;

    ball_t(int newx, int newy, float newsize)
    {
        x = newx;
        y = newy;
        size = newsize;
    }
};

class RecognizeBall
{
    private:
        //Attributes
        byte *yImage; //original y image
        byte *sImage; //y image with sobel filter applied
        int *histogram; //histogram of sImage

        // DEBUGGING
```

```
void FlushIt(); //For poor wireless console
int PrintBloblist(bloblist_t *bloblist);
void PrintHistogram();
void MarkBlobCenter(AdHocList *blobcenterlist);
void FilterWhitePixels();

//utilities
int IsElement(int element, int array[], int *arraylen);
void KillBloblist(bloblist_t *bloblist);
int Fac(int n);
int Combination(int n, int k);
int CountBlobs(bloblist_t *bloblist);

//Image processing
byte* ApplyThreshold(int threshold);
int CalculateThreshold();
int* SmoothenHistogram();
int* MakeHistogram();
void ClearHistogram();
byte* ApplySobelMask();
void ProcessImage();

//Image analysis
float CalcDistanceToBall(int dist);
point_t CalcCenterOfBall(point_t p1, point_t p2, point_t p3);
vector_t CalcVector(point_t p1, point_t p2);
float CalcAngle(vector_t u, vector_t v);
int CheckAngles(point_t p1, point_t p2, point_t p3);
float CalcDistance(point_t p1, point_t p2);
int CheckDistances(point_t p1, point_t p2, point_t p3,
                float *avg);
void MakeTriangles(int array[], int arraylen,
                triangle_t triangles[]);
int AnalyzeImage(AdHocList *blobcenter, ball_t *ball);
AdHocList* CalcBlobCenter(bloblist_t *bloblist);
int CheckPixel(int x, int y);
AdHocList* MakeBlob(int offset);
int CheckPerimeter(int offset);
int FindPotentialBlob(int offset);
int FindNextOnPixel(int offset);
bloblist_t* FindAllBlobs();
```

```
    public:
        RecognizeBall(byte *yImage);
        ~RecognizeBall();
        int* DetectBall();
        byte* SobelImage();
        byte* PaddedSobelImage(byte psImage[]);
        byte* YImage();
        void YImage(byte *yImage);
};

#endif
```

AdHocList.h

```
#ifndef ADHOCLIST_H
#define ADHOCLIST_H
class AdHocList
{
    private:
        struct node_t
        {
            int data;
            node_t *next;
        };

        node_t *first;
        node_t *last;
        int size;

    public:
        AdHocList();
        ~AdHocList();
        void Push(int data);
        int  Pop();
        int  IsEmpty();
        void Print();
        int  Size();
        void Append(AdHocList *data);
        int  Get(int index);
```

```
        int  IsElement(int elem);
        int  Max();
        int  Min();
        int* ToArray(int array[]);
};
#endif
```

AdHocList.cc

```
#include <stdio.h>
#include "AdHocList.h"

AdHocList::AdHocList(void)
{
    first = NULL;
    last = NULL;
    size = 0;
}

AdHocList::~AdHocList(void)
{
    while(first != NULL){
        node_t *tmp = first->next;
        delete first;
        first = tmp;
    }
}

    void
AdHocList::Push(int data)
{
    node_t *newNode = new node_t;
    newNode->data = data;
    newNode->next = NULL;
    if(IsEmpty()){
        first = newNode;
        last = newNode;
    }
    else{
        last->next = newNode;
```

```
        last = newNode;
    }
    size++;
}

    int
AdHocList::Pop()
{
    if(IsEmpty()){
        printf("Error: AdHocList:: Tried to pop an empty list!");
        return -10000;
    }
    int data = first->data;

    if(first == last){
        delete first;
        first = NULL;
        last = NULL;
    }
    else{
        node_t *tmp = first->next;
        delete first;
        first = tmp;
    }
    size--;
    return data;
}

    int
AdHocList::IsEmpty()
{
    if(first == NULL)
        return 1;
    return 0;
}

    void
AdHocList::Print()
{
    node_t *tmp = first;
    printf("\nPRINTING LIST\n");
```

```
    while(tmp != NULL){
        printf("%d ", tmp->data);
        tmp = tmp->next;
    }
    printf("\n\n");
}

    int
AdHocList::Size()
{
    return size;
}

    void
AdHocList::Append(AdHocList *data)
{
    for(int i=0; i<data->Size(); i++)
        Push(data->Get(i));
}

    int
AdHocList::Get(int index)
{
    if(index >= size){
        printf("\nError: AdHocList:: Tried to Get index that don't exist\n"
        return -10000;
    }
    node_t *tmp = first;
    for(int i=0; i<index; i++)
        tmp = tmp->next;
    return tmp->data;
}

    int
AdHocList::IsElement(int elem)
{
    node_t *tmp = first;
    while(tmp != NULL)
        if(tmp->data == elem)
            return 1;
        else
```

```
            tmp = tmp->next;
    return 0;
}

    int
AdHocList::Max()
{
    node_t *tmp = first;
    int Maxval = -1000000;

    while(tmp != NULL){
        if(tmp->data > Maxval)
            Maxval = tmp->data;
        tmp = tmp->next;
    }

    return Maxval;
}

    int
AdHocList::Min()
{
    node_t *tmp = first;
    int Minval = 1000000;

    while(tmp != NULL){
        if(tmp->data < Minval)
            Minval = tmp->data;
        tmp = tmp->next;
    }

    return Minval;
}

    int*
AdHocList::ToArray(int array[])
{
    int cntr=0;
    node_t *tmp = first;
    while(tmp != NULL){
        array[cntr]=tmp->data;
```

```
        tmp = tmp->next;
        cntr++;
    }
    return array;
}
```

Constants.cc

```
#ifndef Constants_H
#define Constants_H

#include <math.h>

//Useful Defines
#define MAX(a,b)   (((a) > (b)) ? (a) : (b))
#define MIN(a,b)   (((a) < (b)) ? (a) : (b))

//Image
#define ORIG_IMAGE_WIDTH   176
#define ORIG_IMAGE_HEIGHT  144
#define SOBEL_IMAGE_WIDTH  175
#define SOBEL_IMAGE_HEIGHT 143
const int ORIG_IMAGE_SIZE  = ORIG_IMAGE_WIDTH * ORIG_IMAGE_HEIGHT;
const int SOBEL_IMAGE_SIZE = SOBEL_IMAGE_WIDTH * SOBEL_IMAGE_HEIGHT;
#define Y_INDEX 0
#define X_INDEX 1

//Histogram
#define NUMBER_OF_GREYS  256
#define K_MIN            -0.2
#define K_MAX            0.02 //not currently used
#define INTERVAL         5 //5
#define SMOOTH_INTERVAL  5 //6

//Blob information
#define MAX_DISTANCE_TO_PIXEL  30 //30 is the lower limit
#define ON_PIXEL        255
#define OFF_PIXEL       0
#define BLOB_PIXEL      128
```

```
#define MAX_BLOB_RADIUS 20 //20 is the lower limit
#define MIN_BLOB_SIZE   6
const int MAX_BLOB_SIZE = (int)(pow((double)MAX_BLOB_RADIUS,(double)2)*M_PI);
#define MAX_NUMBER_OF_BLOBS 20

//Ball recognition
#define DISTANCE_TOLERANCE  3
#define ANGLE_TOLERANCE  10
#define OPTIMAL_ANGLE  60
const float MIN_ANGLE = cos((double)(OPTIMAL_ANGLE - ANGLE_TOLERANCE)*(M_PI/180.0)
const float MAX_ANGLE = cos((double)(OPTIMAL_ANGLE + ANGLE_TOLERANCE)*(M_PI/180.0)
#define SCALE_TO_REAL_WORLD  10/100.0 // Needs testing

#endif
```

Appendix D

C++ Code for the Colour Approach

RecognizeCh1Ball.cc

```
#include "RecognizeCh1Ball.h"

//PRIVATE

/** This fixes the problem when the white ball color merges
  * with the white border
  * */
    int
RecognizeCh1Ball::BallAtBorderFix(BallBlob *ball)
{
    Blob black;
    int xmax = -10000;
    int ymax = -10000;
    int xmin = 10000;
    int ymin = 10000;
    int maxsizex = -10000;
    int maxsizey = -10000;
    int maxnpixel = -10000;

    if(ball->numBlacks < MIN_BLACK_BLOBS)
        return 0;
    for(int i=0; i<ball->numBlacks; i++){
        black = ball->black[i];
        xmax = (black.xmax > xmax) ? black.xmax : xmax;
        ymax = (black.ymax > ymax) ? black.ymax : ymax;
```

```
        xmin = (black.xmin < xmin) ? black.xmin : xmin;
        ymin = (black.ymin < ymin) ? black.ymin : ymin;
        maxsizex = (black.sizex > maxsizex) ?
            black.sizex : maxsizex;
        maxsizey = (black.sizey > maxsizey) ?
            black.sizey : maxsizey;
        maxnpixel = (black.npixel > maxnpixel) ?
            black.npixel : maxnpixel;
    }

    ball->white.sizex = (int)(maxsizex / OPTIMAL_BW_RATIO);
    ball->white.sizey = (int)(maxsizey / OPTIMAL_BW_RATIO);
    ball->white.x = xmin + (int)((xmax - xmin) / 2.0);
    ball->white.y = ymin + (int)((ymax - ymin) / 2.0);
    ball->white.xmax = ball->white.x + (int)(ball->white.sizex / 2.0);
    ball->white.ymax = ball->white.y + (int)(ball->white.sizey / 2.0);
    ball->white.xmin = ball->white.x - (int)(ball->white.sizex / 2.0);
    ball->white.ymin = ball->white.y - (int)(ball->white.sizey / 2.0);
    ball->white.npixel = (int)(maxnpixel / OPTIMAL_BW_RATIO);

    return 1;
}

/** Check how much the black/white ratio condition holds true */
    int
RecognizeCh1Ball::CheckBwRatioCondition(BallBlob *ball,
        double *result)
{
    double wsizex = (double)ball->white.sizex;
    double wsizey = (double)ball->white.sizey;
    double wdiameter = hypot(wsizex, wsizey);
    double ratio = 0.0;

    for(int i=0; i<ball->numBlacks; i++){
        double bsizex = (double)ball->black[i].sizex;
        double bsizey = (double)ball->black[i].sizey;
        double bdiameter = hypot(bsizex, bsizey);
        double tmp = bdiameter / wdiameter;
        if(tmp > ratio)
            ratio = tmp;
    }
```

```
    double min = MIN(ratio, OPTIMAL_BW_RATIO);
    double max = MAX(ratio, OPTIMAL_BW_RATIO);
    *result = (min == max) ? min : (min / max);
    if(*result >= BW_ACCEPTANCE_RATIO)
        return 1;
    return 0;
}

/** Check how much the square condition holds true */
    int
RecognizeCh1Ball::CheckSquareCondition(Blob *white,
        double *result)
{
    double min = (double)MIN(white->sizey, white->sizex);
    double max = (double)MAX(white->sizey, white->sizex);
    double squareRatio = min / max;
    min = MIN(squareRatio, OPTIMAL_SQUARE_RATIO);
    max = MAX(squareRatio, OPTIMAL_SQUARE_RATIO);
    *result = (min == max) ? min : (min / max);
    if(*result >= SQUARE_ACCEPTANCE_RATIO)
        return 1;
    return 0;
}

/** Check if the ball is far away */
    int
RecognizeCh1Ball::CheckSizeCondition(Blob *white)
{
    if(white->sizex > FAR_BALL_THRESHOLD)
        return 0;
    if(white->sizey > FAR_BALL_THRESHOLD)
        return 0;
    return 1;
}

/** Check if the black blob is a true subset of the white blob */
    byte
RecognizeCh1Ball::PartOf(Blob *blackBlob, Blob *whiteBlob)
{
    return (blackBlob->x < whiteBlob->xmax) &&
        (blackBlob->x > whiteBlob->xmin) &&
```

```
        (blackBlob->y < whiteBlob->ymax) &&
        (blackBlob->y > whiteBlob->ymin) &&
        (blackBlob->sizex < whiteBlob->sizex) &&
        (blackBlob->sizey < whiteBlob->sizey);
}

/** Build a ball blob from the white and black blobs */
    void
RecognizeCh1Ball::BuildBallBlob(BlobTable *table,
        Blob *white, BallBlob *ball)
{
    int *numBlacks = &(ball->numBlacks);
    *numBlacks = 0;
    ball->white = *white;
    for(int j=0; j<table->numBlobs[ORANGE]; j++){
        Blob *black = &(table->blobs[ORANGE][j]);
        if(PartOf(black, white)){
            ball->black[*numBlacks] = *black;
            *numBlacks = *numBlacks + 1;
        }
    }
}

// PUBLIC
    int
RecognizeCh1Ball::DetectCh1Ball(BlobTable *table, Blob *white)
{
    BallBlob ball;
    double squareRatio = 0.0;
    double bwRatio = 0.0;

#if USE_FAR_BALL_DETECTION
    if(CheckSizeCondition(white))
        if(CheckSquareCondition(white, &squareRatio)){
            printf("Far ball: square: %f bw: %f\n",
                    squareRatio);
            return 1;
        }
#endif
    BuildBallBlob(table, white, &ball);
    if(CheckSquareCondition(white, &squareRatio) &&
```

```
            CheckBwRatioCondition(&ball, &bwRatio)){
        printf("Close ball: blackblobs: %d square: %f bw: %f\n",
                ball.numBlacks, squareRatio, bwRatio);
        return 1;
    }
#if USE_BALL_AT_BORDER_FIX
    if(BallAtBorderFix(&ball)){
        if(CheckSquareCondition(white, &squareRatio) &&
                CheckBwRatioCondition(&ball, &bwRatio)){
            return 1;
        }
    }
#endif
    return 0;
}

RecognizeCh1Ball::RecognizeCh1Ball()
{

}

RecognizeCh1Ball::~RecognizeCh1Ball()
{

}
```

RecognizeCh1Ball.h

```
#ifndef __RecognizeCh1Ball_H_
#define __RecognizeCh1Ball_H_

#include "../Common.h"
#include "Parameters.h"
#include "Pam.h"

#define USE_BALL_AT_BORDER_FIX 0
#define USE_FAR_BALL_DETECTION 0
const double BW_ACCEPTANCE_RATIO     = 0.8;
```

```
const double SQUARE_ACCEPTANCE_RATIO = 0.8;
const int    MAX_BLACK_BLOBS         = 10;
const int    MIN_BLACK_BLOBS         = 1;
const double OPTIMAL_SQUARE_RATIO    = 1.0;
const double OPTIMAL_BW_RATIO        = 0.25;
const int    FAR_BALL_THRESHOLD      = 7;

class RecognizeCh1Ball
{
    private:
        typedef struct ballblob_t
        {
            Blob white;
            Blob black[MAX_BLACK_BLOBS];
            int numBlacks;
        }BallBlob;
        int BallAtBorderFix(BallBlob *ball);
        int CheckBwRatioCondition(BallBlob *ball, double *result);
        int CheckSquareCondition(Blob *white, double *result);
        int CheckSizeCondition(Blob *white);
        byte PartOf(Blob *blackBlob, Blob *whiteBlob);
        void BuildBallBlob(BlobTable *table, Blob *white, BallBlob *ball);

    public:
        RecognizeCh1Ball();
        ~RecognizeCh1Ball();
        int DetectCh1Ball(BlobTable *table, Blob *blob);
};

#endif
```