THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Visual Object Recognition Using Sony Four-Legged Robots

COMP4911 Thesis Part B Report

Author:
Daniel Chi Kin Lam 2276304
`cklam@cse.unsw.edu.au`

Bachelor of Computer Engineering

Supervisors:

Prof. Claude Sammut
`claude@cse.unsw.edu.au`

Dr. William Uther
`willu@cse.unsw.edu.au`

Submission Date: 17th September 2004

Abstract

Object recognition is an important ability for autonomous robotics agents. This report describes the objects recognition tasks implemented in the rUNSWift 2004 software system, in terms of its background, software architecture, implementation, and problems encountered during development.

This research and development was undertaken as a member of the rUNSWift 2004 team, which represented The University of New South Wales to participate in the Four-Legged League of RoboCup Competition 2004.

Acknowledgements

I would like to give thanks to my team mates in rUNSWift 2004, for their much help and support in all the hard times throughout the year.

I want to give thanks also to Dr. William Uther, our supervisor in the project, who spent many late nights in the lab with us and gave us important advice in all times. Thanks also to Prof. Claude Sammut for his many helpful insights to the project. And also thanks to Brad for his caring on the team throughout the year and in Lisbon.

And special thanks to Min Sub Kim and James Wong, who spent much time chatting with me in the lab and gave me advice on my work. All these have made my life in the lab more enjoyable. Special thanks also to my girlfriend Eleanor, who gave me important support through the mobile phone in my endless sleepless nights in the lab.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of This Thesis

Sometime in the future, we have the following scene happening in an average household,

"Cleo[1], go to the kitchen and hand me the fruit-plate please!"

"Yes, sir." Cleo goes to the kitchen, and then hand to its master a plate.

"Cleo, this is not the fruit-plate. This is for the bread! Go to get the fruit-plate again."

"Yes, sir." Cleo goes to the kitchen, and then hand to its master another plate.

"Cleo, this is for seasoning! Enough. Now go to charge yourself up."

Cleo, our advanced autonomous robot for domestic use, is sent back to its manufacturer in the next day for refund.

Object recognition has been one of the active research areas in AI (Artificial Intelligence) community, especially among AI robotics researchers. For mobile robots to perform useful tasks, it is generally beneficial to have some form of exploration of the environments using vision devices. Some applications are already developed, such as the AIBO entertainment robots produced by Sony which can recognise the face of its owner using its camera, go to the recharge station to recharge itself by recognising the charge-station

---

[1] The inspiration came from Sony's entertainment robot Qrio, but the scenario does not have any implication on the actual robot.

landmark, and perform different tasks base on which command cards are shown to its camera. Some applications are yet to come, the prescribed scenario being one of them.

In this research, we are looking at recognition and distance estimation of specific objects encountered by robots in the Four Legged League of RoboCup competition. They are the beacons, goals, robots and balls on the soccer field. In the rest of this chapter, the background of the competition and the robot soccer team are introduced. In chapter 2, we will look at how this work was previously handled by other teams and by previous rUNSWift teams. In chapter 3 concepts that drive the implementation are explained, followed by the details of implementation in chapter 4 and some problems encountered during the research in chapter 5. And finally, conclusion is drawn in chapter 6.

Because the performance of vision recognition highly depends on many factors like lighting condition and amount of noises in the background, which changes from time to time and vary in different occasions, there is no experimental data presented in this paper, but the problems and suggested solutions are presented instead, to give an idea of the results of the research.

1.2 The RoboCup Competition

1.2.1 Overview

The RoboCup Competition is an international project to promote research in AI, robotics and related areas. Its ultimate goal is "By 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer."

The first RoboCup Competition was held in 1997 in Nagoya, and from then on the number of participating teams has grown from around 40 to over 200 in 2004[2]. The competition is held in around June-July every year in different cities in different countries. In RoboCup 2004 there are five different leagues,

Simulation League

In this league participating teams will write software system that will be loaded into the RoboCup Soccer Simulator, which enable two teams of 11 simulated autonomous soccer players to compete.

---

[2] Counting teams from all leagues.

Small Size League

In Small Size League, robots are built by participating teams to be put in a carpeted soccer field, which is 2.8m long by 2.3m wide. An overhead camera is installed 3m above the soccer field so that global vision is retrieved and fed into an off-field PC for processing. There are five robots in one team and they can connect to the off-field PC to get or transmit required data.

Middle Size League

In Middle Size League, robots are built by participating teams but there is no off-board PC available for data computation during the game and robots need to use their on-board vision.

Four-Legged League

This is very similar to the Middle Size League, except that the robots are not built by participating teams, but specified by the organising committee. In 2004 Sony AIBO ERS-7 and ERS-210/A are allowed.

Humanoid League

In this league, a variety of humanoid bipedal robots are built by participating teams to demonstrate skills that are related to soccer playing. In 2004, four challenges are set for participation — stand on one leg, walk, do a soccer penalty kick, and the free-style event.

The University of New South Wales has participated in the Four-Legged League since 1999 and the name of the team is rUNSWift. It is where this research has come from and base on. The Four-Legged League will be discussed further in the next session.

1.2.2 The Four-Legged League

In The Four-Legged League, participating teams have to use robots specified by the Competition Committee without any modification on its hardware. In 2004 there are choices of either using

1. Sony Entertainment Robot AIBO ERS-210/210A, or
2. Sony Entertainment Robot AIBO ERS-7, or
3. A combination of both in the team

Our team went for choice 2 because of the advanced hardware of the later model. ERS-7 has higher camera resolution, processing power and motor power compare to ERS-210/210A.

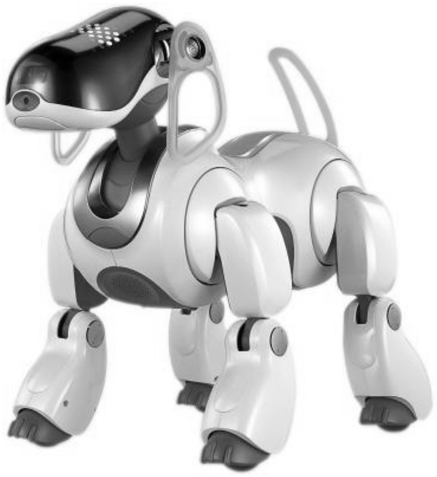


Figure 1.1: Image of Sony AIBO ERS-7

During the game, each robot is fully autonomous with its own sensors, actuators and processors. Wireless cards are equipped with the robots, but they are strictly only used for message passing between teammates, and receiving signals from Robo-GameController to react to penalties and start/finish of the games. Once the game is started, the software has to make use of data provided by the sensors to figure out where it is on the field, where the ball and opponent are, and decide a series of action that maximise the chance of kicking the ball into the opposite goal.

1.2.3 Sony AIBO Robot - Model ERS7

Costing at about the price of an high-end laptop[3], the AIBO ERS-7 has an equivalent amount of delicate computing devices in it. It has 18 joints altogether - 3 joints in the neck, 2 joints in each "shoulder" and 1 joint in the leg, 1 joint to open and close the mouth and 2 degrees of freedom in the tail. With its original design of domestic entertainment, it is also equiped with varying-brightness illuminous face with 28 leds, flipping ears and speaker phone on the chest for interaction with human. Sensors of the robot include

- a 416x320 pixels colour camera[4],

---

[3] Costing $1799 US dollars in August 2004 as shown in Sony US website.

[4] This is the description of the ERS-7 model specification from Sony's OPEN-R website. However, that is only for the Y component in the YUV colour scheme. For U and V, only dimension of 208x160 is available.

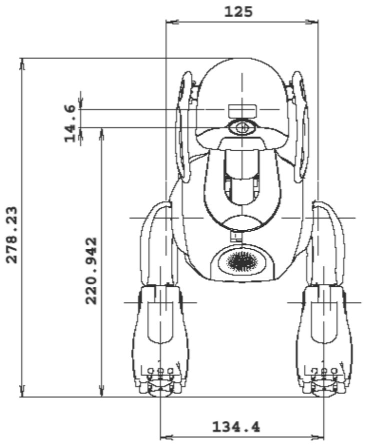


Figure 1.2: Physical Specification of Sony AIBO ERS-7 (Front View, measurements in mm)

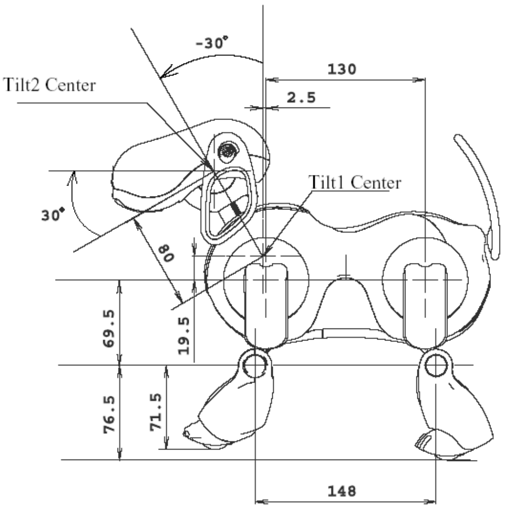


Figure 1.3: Physical Specification of Sony AIBO ERS-7 (Side View, measurements in mm)

- a stereo microphone,
- touch sensors in head and back using electrostaic method,
- touch sensor paw and chin returning on/off values,
- infrared distance sensors with near (5-50cm) and far range (20-150cm) in the head,
- infrared distance sensors on the chest, ranging 10-90cm
- acceleration sensor in 3 axis, and
- vibration sensor

The colour camera is the main source for observing the environment. Touch sensors are used for game controlling and change of states when wireless control is not available. Infrared distance sensors are used in some occasions as an aid to distance estimation.

There are also "hidden" sensors of joint-angle readings and the amount of electric current in all motor joints. For the first time in UNSW rUNSWift team, the joint angle readings are used to calculate the current posture of the robot so that a body-tilt can be added to the head tilt. The amount of electric current in motor joints can be monitored to prevent the disastrous crashing of system current-overloading[5].

1.2.4 The Soccer Field

A soccer field that is used by the Four-Legged League is shown in Figure 1.4. As shown in the diagram, the soccer field has four distinct poles on the corners that help robots to find out where they are on the field. These poles are commonly known as beacons. Something that is not shown in the picture is that the soccer field also has a white barrier surrounding it, to block the views of robots from seeing confusing obstacles lying on the ground outside the soccer field. This barrier has a height of 30cm.

In rUNSWift system and the rest of this paper, the side of the field with blue goal and beacons is known as the "top" side, and therefore the

---

[5] Current-Overloading: This is one of the most difficult stumbling-block encountered by the team in 2004. The robot shuts down itself in a short period of time after game starts, and is originally a safety measure to prevent any human harm caused by motor jam. Up until 2 months before the final RobotCup Competition, the team has to continue development of ERS-7 while robots keep crashing in a few minutes. Even with monitoring the current in motor joints, the problem is only lightly reduced. This problem came to an end when Sony announced a method to "make the jam detection less strict" in the open-r sdk website.

beacons with pink blob in the bottom are known as "left" beacons and those with pink blob on top are known as "right" beacons. This allows the geography in the discussions during development to be easily understood.

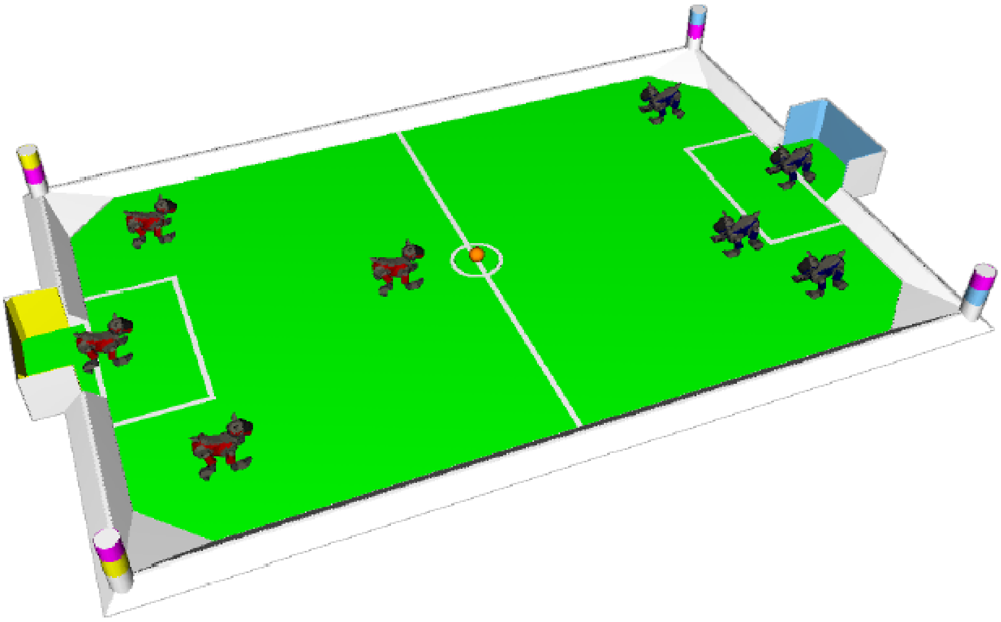


Figure 1.4: The soccer field with robots in kick-off position

All components of the soccer field are colour coded for the robots to recognise, and their size and position are also specified in the Rules of Four-Legged League, to ensure teams from around the world can practice on a very similar soccer field. There are still many calibrations for all teams to work on during the competition, however, because of the varying lighting conditions, slightly different colour coding and having carpets with different thickness.

1.3 System Architecture of rUNSWift 2004

The AIBO robot is a mobile robotics agent that acts on the environment. On the highest level, we have rUNSWift system working through the OPEN-R operating system to receive sensor information and act through actuators to the environment, as shown in Figure 1.5.

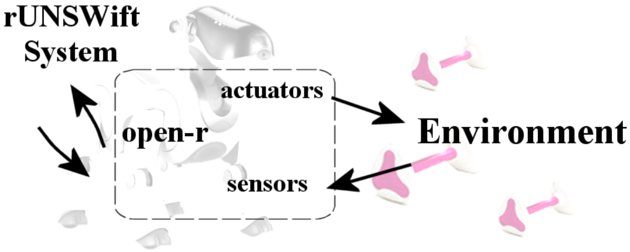


Figure 1.5: How rUNSWift system interacts with the environment

Internally inside rUNSWift, there are behaviour, localisation and vision modules bundled in an operating system specific object called an APERIOS object. There are also locomotion module locating inside actuator control object and wireless module locating inside wireless object, as shown in Figure 1.6.

In 2004 the infrastructure of rUNSWift has not been modified, except for the behaviour module where most implementations have been ported from C++ codes to Python codes. This helps to shorten development time spent in fine-tuning behaviours.

1.4 Changes from 2003 to 2004

1.4.1 Changes of Rules

In 2004, new rules are introduced to penalise robots that run into one another in unnecessary conditions. In particular, the most significant one relating to robot recognition being the Field Player Pushing Rule[6],

> "Any robot pushing another robot for more than 3 seconds will be removed from play for 30 seconds as per the standard removal penalty. The closest robot (including the goal keeper) to the ball on each team, if it is within 2 dog-lengths of the ball, cannot be called for pushing."

This rule demands a higher requirement of accuracy in recognising robots and knowing how far they are from the observing robots.

---

[6] In section 4.7 of Sony Four Legged Robot Football League Rule Book 2004.

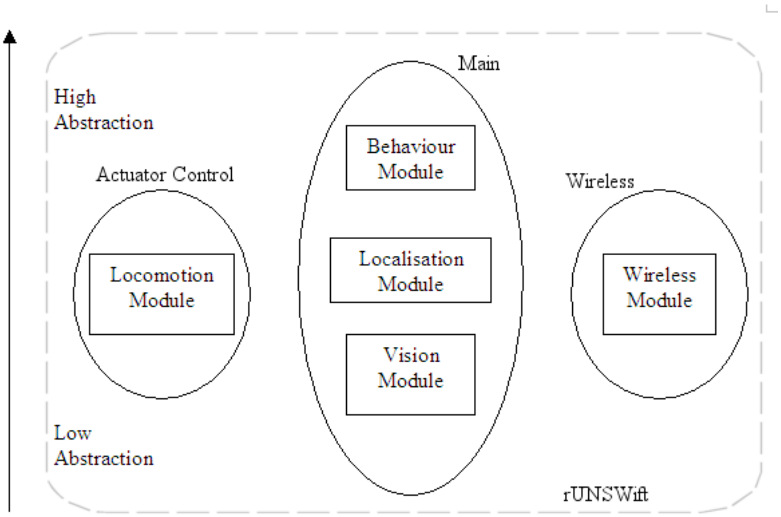


Figure 1.6: Internal Architecture of rUNSWift 2004

1.4.2 Changes of Field

The soccer field remains unchanged compared to 2003 except for two modifications.

Firstly, the middle 2 beacons are removed from the field. This poses great challenge for better vision and localisation systems, as the robot will have less chance of seeing a landmark, and it needs to recognise and estimate the distances of far-away beacons more than before to compensate the loss of landmarks.

Secondly, the height of barrier surrounding the soccer field was cut down by half. In 2003 this barrier had a height of 60cm and in 2004 it was cut to 30cm. The effect of this change is two fold. First of all, the removed white barrier will introduce more area that contains different colour of small blobs, and this consumes extra computation power because these noises will be segmented and formed as a blob. Also, more noises mean that the chance of getting false-positive[7] is increased - new filtering conditions need to be introduced. In particular, the beacons will no longer be fully covered by the barrier and they may be merged with the background if there is a similar colour beside it.

---

[7] False-positive means classifying an image to be an object while the object is not there.

Chapter 2

Previous Work of High-Level Vision Recognition

Object recognitions done by rUNSWift in previous years have been very similar what the current system is having, i.e. building sanity checks to filter out undesired blobs. There have been some attempts on object recognition using the edges of the object, done by Raymond Sheh in 2003 to recognise the round shape of soccer ball, but the algorithm was too slow to be applied in competition condition.

German team, who won the World Championship of Four-Legged League in 2004, used line scanning to determine features of objects. Their way of object recognition tends to have more low-level processing operations, and concentrates on different objects using different techniques.

Some other teams participating in RoboCup use contour detection, but no team did this work well enough to gain advantages from it.

Chapter 3

Principles and Concepts

Before the details of implementation are discussed in this paper, this chapter will first go through the infrastructure of the vision module in rUNSWift system. Also, the concepts of sanity checks and visual objects are central to object recognition, and will therefore be explained in this chapter as well.

3.1 Vision Module Overview

To some extent, the vision module is very isolated from the rest of the RoboCup system because all that the other modules of the system want to ask from vision are “Which objects are recognised?”, “Where are these objects on the soccer field?”, and in case of recognising robots, “Which way are they facing?”. Therefore, one can consider the Vision Module as a black box of which the input is an array of YUV values coming from the camera at a rate of 30 frames/s, and the output is the details of any recognised objects. This section will briefly explain the processes that occur inside this black box (fig. 3.1)[1].

3.1.1 Colour Classification

The first input that is fed into the Vision Module is an array of values that represents the colour of pixels of the current image grabbed by the video camera. This array of values specifies the colour of the image using YUV colour scheme, a colour scheme that is commonly used in television, in which colour can be specified by combining the luminance component Y and the

---

[1] The readers should be reminded of the fact that although the whole vision module is briefly gone through here, the author actually implemented only the object recognitions. The vision module is explained in this section for the readers to easily understand subsequent chapters.

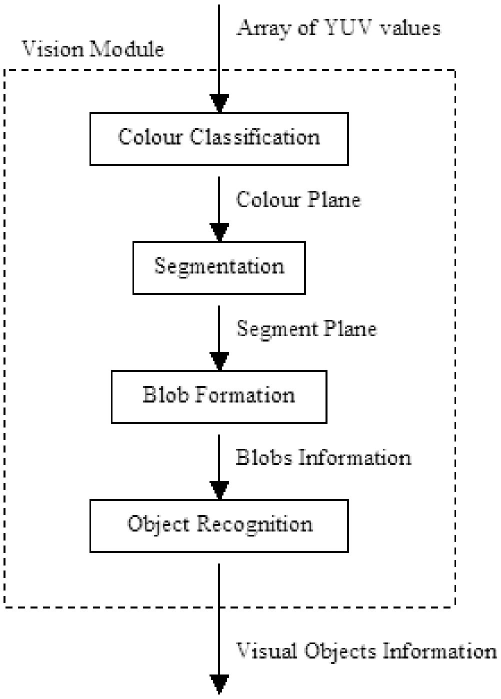


Figure 3.1: The Internal Components of the Vision Module

chrominance (colour) components U and V[2]. To be exact, each image has a dimension of 416 x 320 pixels if we consider only the Y component, and the "real" dimension is 208 x 160 pixels. Why is there a difference? This is because the camera provides more information on Y values compared to U and V values. A graphical explanation of this configuration is given in fig. 3.2,

The array is fed into the RoboCup system at a rate of 30 frames per second. This constrains the system to process the input within 1/30 = 0.25 second per frame. Because of this efficiency requirement, the team decided to process the image with the lower dimension of 208x160 where each pixel has all Y, U and V components, while the extra 3 Y components in each pixel is ignored. The other reason of this decision is that the Y component is responsible for brightness of the colour, if this Y component is changed but the U and V components remain unchanged, the colour classification is

---

[2] In case of this particular YUV scheme provided by ERS-7, each component can have a value of [0, 255], so the maximum number of colour that can be represented is 256x256x256 = 16,777,216 colours

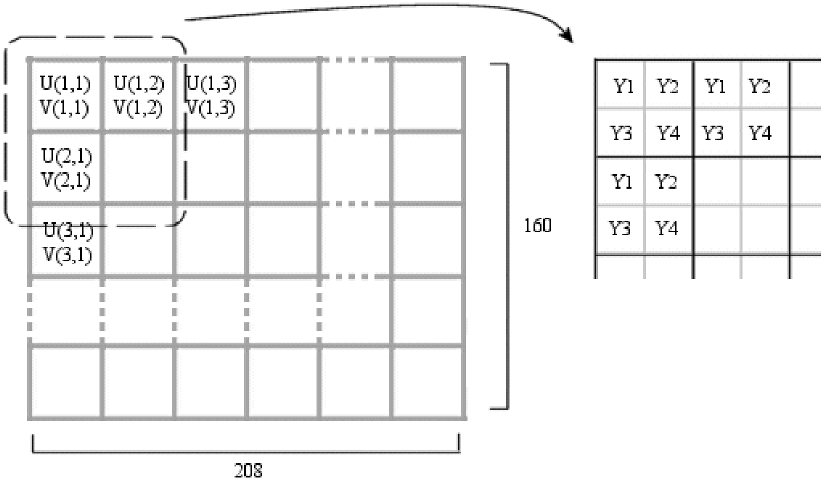


Figure 3.2: YUV Information provided by ERS-7

not expected to change a lot.

To classify this array of values into colour labels that are useful to the rest of the RoboCup system, a three dimensional colour look-up table is built using machine learning technique, so that given any raw values from each of the Y, U and V component, a corresponding colour can be decided. A full colour look-up table that can map every colour in the ERS-7 YUV scheme would require 256x256x256 = 16,777,216 bytes = 16 megabytes[3] if we use 1 byte to store each colour label. However, because of storage limitation, a smaller table that simplify each colour component from 256 values to 128 values is used instead, so that only 128x128x128= 2,097,152 bytes = 2 megabytes is required.

After consulting the colour look-up table for each pixel, 208x160 colour labels would be stored in an array for other parts of the rUNSWift system to use. This array is commonly known as CPlane (Colour Plane) in rUNSWift system and in the rest of this paper.

3.1.2 Segmentation

A segment is a group of pixels that have the same colour and are joined together side by side. Therefore, for each line of the image there is at least

---

[3] The value 256 comes from the fact that each Y, U, V component has 256 values ([0, 255]).

one segment, and there is at least 160 segments in one image. An example is shown in fig. 3.3, in which a small part of CPlane with only 2 colour is shown, and the segments of the darker colour are labelled.

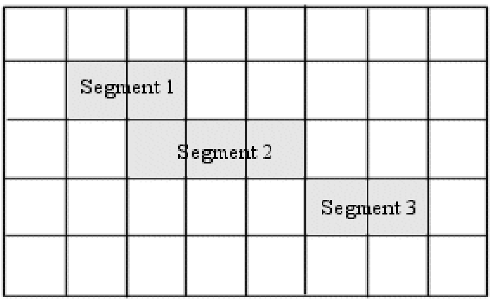


Figure 3.3: Sample Segments formed after Segmentation (only the darker colour)

Although fig. 3.1 shows that the segmentation process occurs after the colour classification, this presentation is actually for easier understanding of the overall process. In the implementation segmentation and colour classification actually occur at the same time to save processing time, because then there is no need to go through the whole CPlane a second time.

3.1.3 Blob Formation

After the segment information is collected, all segments that are connected to each other and with the same colour are joined together to form a group, commonly known as blobs in the rUNSWift system.

In 2004 rUNSWift has tried a new definition of "connected" segments. In previous years, two segments are interpreted as connected whenever there is at least one pair of pixels where one pixel is right on top of the other. This is called 4-way joining because a blob can be extended if a new pixel is located on top, left, right or bottom of the original blob.

A slightly different sense of connection is experimented in 2004, in which a blob can be extended if a new pixel is located not only on the 4 directions, but also the diagonal top-left, top-right, bottom-left, bottom-right directions. This is called 8-way joining. If we look at fig. 3.3, segment 1, 2 and 3 will all be joined together and form a blob if 8-way joining is used, while only segment 1 and 2 will be joined and form one blob, leaving segment 3 as another blob if 4-way joining is used. It was decided not to use the 8-way joining algorithm in the game condition, however, because experimental results showed that 8-way joining often connect noises together and created undesired large blobs.

Statistics about each blob are collected in this blob formation process. The main attributes that are used by object recognition are listed in Table 3.1. Notice that there is no information about the shape of the blob, and if object recognition is desperate to know about it, it needs to work it out from the CPlane itself.

Attribute	Description
colour	The colour of the blob
area	The number of pixels in the blob
min/max x	The minimum/maximum x-coordinate reached by the blob
min/max y	The minimum/maximum y-coordinate reached by the blob
cx	The x-coordinate of centroid of the blob
cy	The y-coordinate of centroid of the blob

Table 3.1: Main Attributes of Blobs Used in Object Recognition

3.1.4 Object Recognition

With all the blob information collected, further investigation can be carried out to determine whether each blob or a few blobs together is a particular object on the field. These checks are called sanity checks and are what the entire object recognition based on.

After different sanity checks are run, objects will be recognised together with their properties. This information will be available to other modules in the system — Localisation module can update the current position of the robot if any landmark is observed, behaviour module can produce the most rational strategy base on where the ball, team mates, opponents and the goal are, and the wireless module can send object information to the wireless base station or team mates if it is needed.

3.2 Sanity Check

Sanity check is a termed used in rUNSWift system that refers to the filtering conditions of blobs that are not real objects.

For example, in order to recognise a ball on the field, the object recognition process will go through all the blobs that are in orange colour and ask questions like "Is this blob too thin (its height less than 3 pixels)?", "Is this blob too small (its area less than 5 pixels)?" to determine whether a blob is a valid ball.

3.2.1 Threshold Values of Sanity Checks

From time to time, sanity checks are revised to make sure the threshold values used in the checks are optimal in maintaining a fine balance between accepting too many blobs that are not objects to be objects (called false-positives) and failing to recognise blobs that belong to real objects as objects (called false-negatives). This optimal fine-tuning of threshold values is not independent inside object recognition, but also affected by implementation of lower and higher level modules.

In higher-level modules, take localisation module as an example, it is known that a false beacon will seriously tear apart the localisation of the robot. Therefore it would be important to maintain a low level of false-negatives, even though that might mean missing some real beacons. Similar consideration applies for recognising goals, balls and robots, in which knowledge from strategy module might favour seeing or not seeing objects under some conditions.

In low-level vision processing, on the other hand, the colour calibration can favour the amount of a particular colour to be seen, which will then affect the threshold values in sanity checks.

Therefore, it is important to have plenty of communication among the team members to ensure such knowledge is shared.

3.2.2 The Ordering of Sanity Checks

The ordering of sanity checks is base on two factors.

1. If this sanity check can filter out most of the false objects
2. If this sanity check is quick to execute

If the answer to these questions are both positive, then a particular sanity check should be run first. The motivation of this prioritising is that if a blob is not a real object, it is desirable to move on to the other blobs quickly. Thus if there is a sanity check that can efficiently filter out most blobs, then it should be put in front.

3.3 Visual Objects

For rUNSWift 2004, there are four kinds of objects on the soccer field that we want the robot to recognise, they are

- Four distinct beacons
- Yellow and blue goals
- Other robots
- The soccer ball

After the object recognition process, the vision module should be able to tell other modules about where these object are on the soccer field. To meet this objective, the vision module has an array of 17 Visual Objects that is readable by other modules. The indexing of the array and explanation of the important attributes of each Visual Object are shown in Table 3.2 and 3.3 respectively[4].

Notice that there are six beacons instead of four beacons in the array. This is only for backward-capability purpose since there were green beacons in the middle of the field in 2003. In the implementation, there is no attempt to recognise these beacons in the rUNSWift 2004 system.

Thus, the primary goal of object recognition process is to fill in necessary contents in this array for each processed frame. Note that although there are many attributes in the Visual Object class in VisualCortex.h, the only important ones that matter from the perspective of other modules are those that are listed in Table 3.3.

---

[4] Further details can be found in VisualObject class in VisualCortex.h

Visual Object Array Index	Visual Object Name
0	Ball
1	Blue Goal
2	Yellow Goal
3	Blue Left Beacon
4	Blue Right Beacon
5	Green Left Beacon
6	Green Right Beacon
7	Yellow Left Beacon
8	Yellow Right Beacon
9	Red Dog 1
10	Red Dog 2
11	Red Dog 3
12	Red Dog 4
13	Blue Dog 1
14	Blue Dog 2
15	Blue Dog 3
16	Blue Dog 4

Table 3.2: The Array of Visual Objects

Attribute	Description
d	Distance of object from the observing robot
h	Heading angle in degree indicating how deviated the object is from the observing robot horizontally
elev	Elevation angle in degree indicating how high the object is above ground
cf	Confidence factor, a value to indicate how certain the vision module can say about the object being observed
var	Variance of location of object, functions like cf
angleVar	Variance of heading angle of object

Table 3.3: Important Attributes of a Visual Object

Chapter 4

Details of Implementation

This chapter covers the details of implementation of object recognition in rUNSWift 2004. The first section gives an overview of object recognition in terms of which objects are recognised first and the reason behind that. The second section describes some special features used in sanity checks that would better be made clear in the beginning, before going into specifics of recognitions of different objects in the following sections. The last section provides an overview of debugging techniques used in the development of object recognition.

4.1 Implementation Overview

After blobs information is collected, object recognition can proceed. At the highest level, the algorithm looks like codes in Figure 4.1. Note that this outline and other pseudo-codes listed in this paper are only shown here to make the discussions easier to understand, thus some details from the real implementation are omitted.

Landmarks are looked for first because any observed landmarks provide good geographic information that the sanity checks of other objects can base on. For example, if a beacon is observed, it is not possible to have other objects observed above it. Beacon check also needs to come before the goal check. Imagine if goal check came first, and a blue blob is being assessed to be goal or not. It is then necessary to check if there are surrounding blobs that would make this blob a beacon. Therefore the beacons are checked first.

The aim of BeaconsSanities() and LandmarkSanities() functions are to check if the observed beacons and goals are consistent with one another. For example, it would not make sense to see a blue beacon and the yellow goal at the same time. The false object needs to be removed before any further recognition of other objects.

```
FindVisualObjects() {

    FindBeacons();      // Find all the beacons.
    BeaconsSanities();

    FindGoal();         // Find the blue and yellow goals.
    LandmarkSanities();

    FindBall();         // Find the ball.
    FindRobot();        // Find red and blue robots.
}
```

Figure 4.1: Pseudo Codes - FindVisualObject()

After checking the landmarks, balls and robots search are carried out. Traditionally, it is better to look for ball first, because when a ball is very close to the camera of the robot, or when the ball is under dark shadow, it will turn red and could be classified as robots. Therefore it is useful to check the existence of the ball first, and then rule out the possibility of robots under some conditions.

However, it was mention this arrangement is “traditional” because for ERS-7 in 2004, not only does the system need to rule out possibilities of seeing robots from seeing ball, but also the other way round. Because there are often orange blobs on the red robots, the system needs to rule out the possibilities of seeing ball given the red robots are seen under some conditions. Thus the order of checking ball first or checking robots first is not so important any more.

4.2 Special Features Used in Sanity Checks

Before looking at the details of recognition of different objects, this section will first go through the details of some sanity checks that are used in different object recognitions.

4.2.1 Horizon Check

Horizon line is a line that can be drawn on the CPlane to indicate the height level of the camera. It is developed in 2004 and calculated using the pan, tilt, crane angles of the head and the length between the tilt-base of the neck to the ground. An example is shown in Figure 4.2, in which the horizon line is drawn in black colour, and a pink line is drawn below it to indicate the direction of the ground. The lines are drawn by the OffVision tool for debugging purpose.

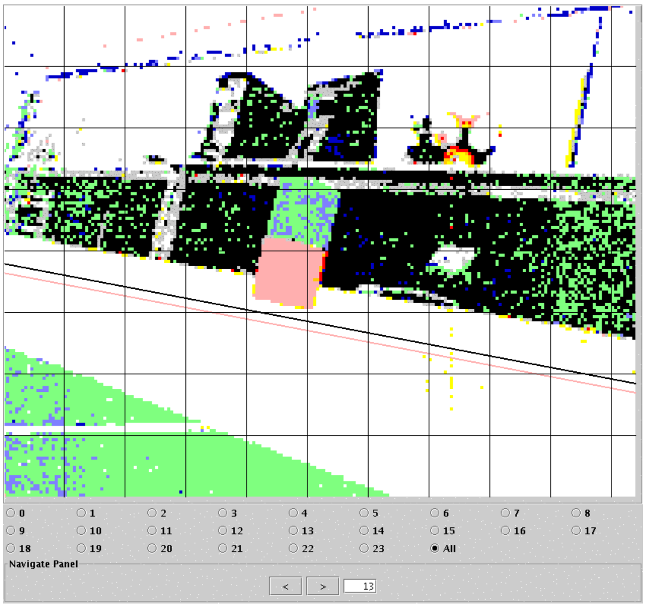


Figure 4.2: The Horizon Line

Unfortunately, the accuracy of horizon is changing all the time due to head movement and the body movement of the robot. When the robot is walking fast, the tilt angle of its body is also changing quickly and this adds noise to the horizon line calculation. Kim Cuong Pham later came up with a way to calculate the tilt angle of the body of the robot using the values of the motor joints, so that the horizon line calculation can take it into account.

4.3 Beacons

As shown in Section 4.1, the recognition of beacons consists of first finding the beacons, then check if there is any inconsistency between them. These two steps will be discussed in this section, followed by a discussion on how beacon distance is estimated.

4.3.1 Beacon Recognition

Because beacon is so important to the localisation of the robot, and localisation is so important to building a reliable strategy — it is very important to get the beacons recognised correctly. In addition to this, the beacon is also the only object that is composed of multiple blobs in multiple colours. This has made beacon recognition the most complex among all object recognitions.

Details of FindBeacons()

The highest level of beacon recognition is implemented in the FindBeacons function. An outline of this function is shown in Figure 4.3.

```
Go through each pink blob...

    Use FormBeaconFromPink to find if any BLUE beacon can be formed.
    Use FormBeaconFromPink to find if any YELLOW beacon can be formed.

    Check if 2 beacons are formed on 1 pink blob
      Y -> Choose only 1 beacon, ignore the other one

   Use elevation to see if pink blob is on top or bottom,
    thus deduce if it is a left or right beacon

    If this beacon has already been observed,
        Ignore the later beacon
    Else
        Store this beacon
```

Figure 4.3: Pseudo Codes - FindBeacons()

This function is not very complex, because most complicated sanity checks are implemented in the FormBeaconFromPink function. This FormBeaconFromPink function will try to match all the blue blobs or all the yellow blobs with the current pink blob of the loop to see if a potential beacon can be formed. It will be discussed in detail below.

Sometimes it happens that FormBeaconFromPink will generate 2 beacons that share the same pink blob. This is certainly impossible. Therefore one must be false and needs to be eliminated. To decide which of the beacon is to be thrown out, two methods were experimented during development.

Method 1 The first method compare the ratio of pink blob area to other colour blob area of the two beacons, and keeps the beacon that that has a ratio closer to 1.The motivation here is to hope that a correct beacon would have an equal amount of blob area for both colour, while the false beacon would not.

Method 2 The second method compares the elevation of the two beacons, and simply keep the one that has a lower elevation. The motivation here is that most false beacons are formed by noises in the background, but rarely are false beacons formed on the soccer field. So the correct real beacon must be the one that is closer to the ground, assuming the false one is flying in the air.

Both methods have their advantages and downsides. For method 1, there are times when the real beacon does not have a good proportion between the areas of its two blobs, e.g. when the robot is under poor lighting condition. In particular, beacons are only partly observable when it is in the corner of the image. For method 2, the assumption that there are no false beacons formed on the soccer field could be wrong too. Considering the pros and cons of these two methods, method 2 was chosen at the end because beacons are often seen in corners, which breaks method 1 too often.

The beacons found from FormBeaconFromPink function have no distinction between left and right beacons. Since the only difference between left and right beacons is the left one has the pink blob on top while the other has pink blob in the bottom, therefore in line 9, the codes check whether the pink blob of the beacon is closer to the ground or the other blob is closer to the ground to determine this.

Sometimes the same beacon is observed twice. Again, one would be false and needs to be thrown away. This is done in line 12 in the pseudocodes in Figure 4.3. However, there is no special method here to determine which one to throw away. The codes will simply throw away the ones that come later. This is because our limited time is limited, and it is not possible to implement everything we want, but to choose to tackle problems that would really make a difference if they were solved. A good indication of importance of a problem is the number of samples that are misclassified because of it. In this case, not much harm could be observed even though there is no special method is implemented in line 12. Therefore development effort is put into other areas instead.

After the above checks, beacons would be confirmed and saved in the Visual Object array.

Details of FormBeaconFromPink()

An outline of FormBeaconFromPink is shown in figure 4.4. This function is called for each of the pink blob in the image and for each of blue and yellow colour. Some details of this function are taken out from it because otherwise the figure would become too big to be displayed.

```
FormBeaconFromPink(Pink Blob, Match Blue/Yellow) {

    Is the whole CPlane below horizon?

    // Sanity checks for the pink blob.
    Is pink blob too high above horizon?
    ...

    Go through each blob of blue/yellow {

        // Sanity checks for the other blob.
        Is this blob already formed as another beacon?
        ...

        // Sanity checks for the relationships between two blobs.
        Are the two blobs close to each other?
        ...

        // Adjustments for centroids of blobs.
        Adjust the centroid of the pink blob (disabled)

        If the beacon is partially seen,
            Re-calculate the centroid of the partially seen blob

        If the beacon is merged with background, (disabled)
            Re-calculate the centroid of the merged blob

        // Final sanity checks.
        Is the aspect ratio of bounding box not acceptable?
        Is the colour below the lower blob correct?

        Estimate the distance to the beacon,
        Is the calculated height too big/small?

        Save this beacon
    }
}
```

Figure 4.4: Pseudo Codes - FormBeaconFromPink()

The first thing to check is whether the whole CPlane is below the horizon line (details on horizon line can be found in Section 4.2.1). If it is, then the robot must be looking down and it is not possible to observe any beacons, and the function can return straight away. Because it is such a quick check that does not even require information on blobs, it is put on the very beginning of the function. In fact, it could be put in the beginning of FindBeacons function to save the time to run the pink-blob for-loop, although the difference would not be much.

Then, the function will check if the input pink blob could possibly form a beacon. The questions that are asked in line 6 and 7 are listed below (in this order):

1. **Is pink blob too high above or too low below horizon line?**
   While the very first sanity check checks if the whole CPlane is below the horizon, this check will check how much the pink blob is deviated to the horizon line on CPlane, if there is one.
2. **Is the bounding box of pink blob too small?**
   The bounding box refers to a rectangular box that can fully contain the blob. If this box is too flat or too thin, then it is likely to be a background noise.
3. **Is the solid ratio of the pink blob too small?**
   This maybe easier understood as density. It is simply the area of the blob divided by the area of the bounding box. If this value is too small, that means the blob is very scattered, or has a diagonal orientation. In both cases the blob would not be part of a beacon.
4. **Is the bounding box aspect ratio too big?**
   This is the longer side of the bounding box divided by its shorter side. If it is much larger than 1, it means the shape of the blob is far from a square, which would then not be part of a beacon.

A special case occurs for solid ratio check and bounding box aspect ratio check when a pink blob is lying on the edge of an image, it would not look like a square. This case is special checked for and catered in the code.

In line 9, after checking the properties of the pink blob, the function will then go through every blob with blue or yellow colour (depending on the input) to see if the pink blob and the other blob together compose a beacon. Similar sanity checks for the blue or yellow blob like the ones above are run, and they are shown in the list below:

1. Is this blob already formed as another beacon?

2. Is the bounding box of the other blob too small?
3. Is the solid ratio too small?
4. Is the bounding box aspect ratio too big?

The first thing to check here is if this blob has already been formed as another beacon. If it has, we would not like to overwrite the beacon information. The rest of the sanity checks have already been discussed, thus they will not be repeated here.

Next, in line 16, it comes to the more complicated checks that concern the relationship between the two blobs.

1. **Are the two blobs close to each other?**
   If two blobs are not joined together side by side, they cannot belong to the same beacon. To determine this, a centroid-to-centroid distance is first calculated using the centre of both blobs. Then the radii of both blobs are calculated by counting the number of pixels between a blob and its bounding box along the line that connects two radii.

2. **Is the distance between two blobs too big?**
   This check looks at the centroid-to-centroid distance mentioned above. If it is too big, it is not possible to be a beacon because a real beacon, no matter how close it is to the camera, has an upper limit.

3. **Is this beacon vertical to the ground?**
   A real beacon must be vertical to the ground. This check calculates the angle between the line joined by the two blobs and the horizon line to check if they are about 90 degrees apart.

4. **Is the beacon elevation too small?**
   The elevation angle to a real beacon may be very large when the robot is close to the beacon, but it has a lower bound because the robot is below the beacon at all times.

5. **Is the beacon above or below horizon line a lot?**
   This is similar to checking the blobs against horizon line mentioned earlier in this section, except that this time the middle bottom point of the beacon is used for checking. This will make noises of blobs that are below the horizon easier to be catch. A CPlane that shows the location of the mid-bottom point of the beacon using an arrow is shown in Figure fig-mid-bot-point.

In 2004, it was noticed that the pink blob of a beacon often appear orange or red. This problem seriously affected beacon recognition when an

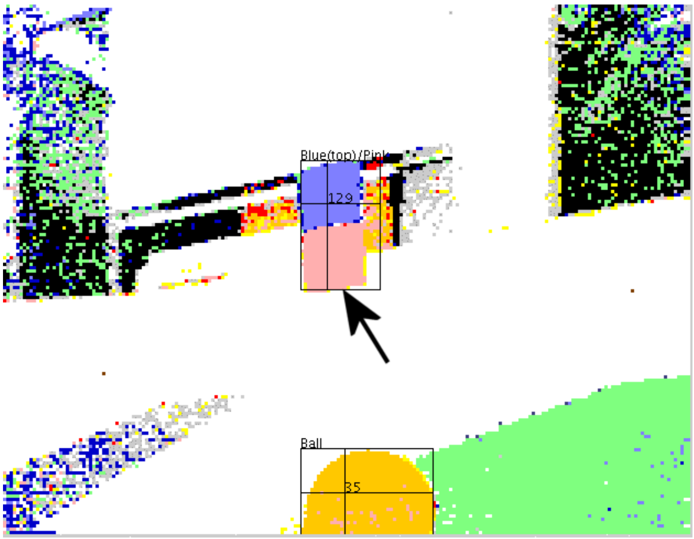


Figure 4.5: The Middle-Bottom point of Beacon

observed beacon is far away, in which a great proportion of the original pink blob would turn red or orange. This problem is mainly due to a chromatic distortion of the camera, which will be discussed in Section 5.2. Because it takes time to develop solutions in low-level vision processing, an attempt is made in higher-level recognition to help with this problem, which takes place in line 20 in the listed codes of FormBeaconFromPink function. Basically it tries to check if there are red or orange pixels around the pink blob, and equivalent these pixels as pink in re-calculating the centroid of the blob. In a later stage of the development, it is found that this centroid readjustment is often over-correcting the good blobs as well. Therefore the re-adjustment is cancelled.

In line 22, a correction to the centroid is made for any blob that is on the edge of CPlane. This is necessary because otherwise the centroid-to-centroid length would be incorrect and lead to a wrong estimation of distance to beacon.

In line 25, an attempt was made to correct the centroid of a blob when it is merged with the background. This could happen because the barrier of the soccer field can no longer block the background noise from the beacon. If a person is standing behind a beacon and wears blue jeans that is classified as the beacon blue, then the blue blob of the blue beacon can merge with the jeans to form a bigger blue blob. To correct this undesirable scenario, the merged blob is simply ignored, and the information of the beacon is reconstructed using only the smaller blob. In the beginning this attempt was quite successful, helping the robot to recognise beacons even when one blob of the beacon is completely disrupted by the background. However, this correction has to be disabled at the end, because the correction is too easily triggered, and it often generates centroid-to-centroid length that is not correct because the smaller blob that this method relies on is often noisy.

The final few sanity checks are now ready to be called. They are at this very end of the function because they rely on the correct bounding box and centroids.

1. **Is the aspect ratio of bounding box not acceptable?**
   This is similar to the earlier bounding box aspect ratio check in this section, except that this time a bounding box that contain both blobs are constructed. If the aspect ratio of this combined bounding box is not right, the blobs could not form a beacon.
2. **Is the colour below the lower blob correct?**
   This is a good property of beacon to be used in sanity check. A rectangular box is defined below the potential beacon, and the number of white, grey, black, green, blue and red colour pixels inside the box are counted. Different threshold values can then be checked on this statistics. It is not likely to be a beacon if this area has no white at all, or is full of green, for example.
3. **Is the calculated height using distance too big/small?**
   Distance between the robot and the beacon is estimated using a linear function. The details of this estimation will be discussed in Section 4.3.3. With the distance, the function can then calculate the height of the beacon using the distance and the elevation angle. Since the beacon is static at all times, this sanity check can filter out false beacon that has ridiculous center-to-center distance and therefore calculated height.

After all these sanity checks, the pink blob and the other blob are saved as a potential beacon and is subject to more sanity checks in FindBeacons and BeaconSanities functions.

4.3.2 Internal Checks Among Recognised Beacons

The FindBeacons function will find out all the beacons that are rational by themselves. Then it comes to BeaconsSanities function in which their relationship is examined and filter out those that do not make sense. An outline of pseudo-codes of BeaconsSanities function is shown in Figure 4.6.

```
BeaconsSanities() {

    If only see one beacon,
        Return

    Else if there are three beacons
        Throw out the one outside the pair in same colour

    Else if there are four beacons
        Choose the pair which better distance.

    Else if there are two beacons

        Is the right beacon in the left of left beacon
            OR the left beacon in the right of right beacon?
        Are beacons in opposite sides?
        Is one much higher than the other?
        Do their sizes differ a lot?

        If Y any above,
            Choose the one lower in elevation.
}
```

Figure 4.6: Pseudo Codes - BeaconsSanities()

When there is just one beacon, there is no other beacon to check against and therefore the function can just return.

When there are three beacons, it is either one of these two combinations:

1. One blue beacons and two yellow beacons, or
2. One yellow beacon and two blue beacons

It is impossible to see beacon in opposite sides at the same time, therefore the beacon that is not in the pair will be thrown away.

When there are four beacons, one of the pair of beacons must be false. It is observed that whenever a robot on the soccer field observes two real beacons, the distance between these two beacons is approximately a constant. Therefore in this sanity check, the pair that has a distance closer to the ideal distance will stay, while the other pair will be thrown away.

When there are only two beacons, there are still more checks to do. If the right beacon is observed on the left hand side, or vice versa, one beacon must be false. If a blue beacon and yellow beacon are observed at the same time, one must be wrong. Even when both beacons have the same colour, and are arranged in the right order, it is still important to check if their height and sizes are similar.

After these logical checks, the beacons are confirmed and they will not be filtered out in other parts of the object recognition system.

4.3.3 Beacon Distance Estimation

The rUNSWift system estimates the distance between the robot and beacon using the fact that beacon distance is inversely proportional to the centroid-to-centroid distance of beacon blobs. Base on this relationship (4.1), constants of the formula shown in (4.2) are found using experimental results. A diagram in Figure 4.7 also shows what it meant by interCentroidDistance, also known as the centroid-to-centroid distance.

$$beaconDistance \propto \frac{1}{interCentroidDistance} \tag{4.1}$$

$$beaconDistance = \frac{c1}{interCentroidDistance} + c2 \tag{4.2}$$

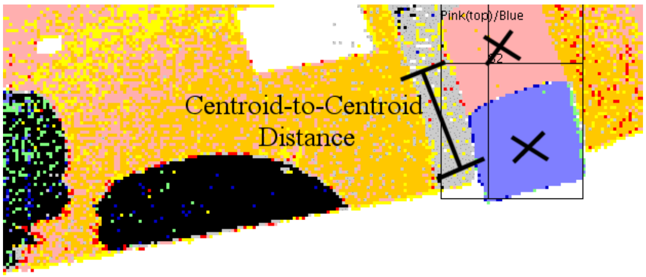


Figure 4.7: The Centroid-to-Centroid Distance of a Beacon

To find the constant c1 and c2, the robot is placed in the soccer field at recorded distance to the beacon and is set to look at the beacon so that beacon centroid-to-centroid distance can be recorded as well. A linear regression line is then drawn on experimental data points, having beacon distance against (1/centroid-to-centroid distance). The slope of this regression line will then be our required c1 and the y-axis offset is c2.

After a basic value is obtained, it is not necessary to run the experiment again to get all the measurements for calibration of a new environment. Instead, the robot can be placed on the field at specific beacon distance, and a person can change the beacon constants of the robot through wireless commands and check the resultant beacon distance without knowing the centroid-to-centroid distance.

4.4 Goals

Similar to beacon recognition, goal recognition is also composed of looking for goals in the image, then check if there are inconsistencies between the recognised objects. The algorithms used will be discussed in detail below.

4.4.1 Goal Recognition

The goal recognition is simpler compare to beacon recognition because only one colour is needed for consideration. One the other hand it is also more difficult because robots on the soccer field can be blocking the view to a goal, then the goal would be broken down to several blobs.

In 2004 a difficult problem faced by goal recognition arouse because of the reflexive texture of the “white skin” of ERS-7. This shinny outer layer of the robot often reflects colour of its surrounding object. Therefore yellowish colour often appears on robot body when a robot is holding a ball. Lots of fine-tuning effort was made to filter out these undesirable yellow blobs, while still recognising yellow goals correctly.

Details of FindGoal()

The FindGoal() function is called by FindVisualObject() to collect information of all observable goals. It is a short function that delegates the task to the “real” function that performs sanity checks and practices code reuse by specifying which colour of goal to look for. An outline of the function is shown in Figure 4.8.

Beside the colour of goal to look for, the input also contains the beacons that are expected to appear in the left hand side and right hand side of the goal. This is provided for the sanity checks to use. Note that the order of input beacons to LookForColourGoal() function is different for blue goal and yellow goal. This is not a mistake — it is because of our notion of left and right is different from the view point of the robot in the soccer field.

```
FindGoal() {

    LookForColourGoal(Blue, Blue Left Beacon, Blue Right Beacon)
    LookForColourGoal(Yellow, Yellow Right Beacon, Yellow Left Beacon)

}
```

Figure 4.8: Pseudo Codes - FindGoal()

Details of LookForColourGoal()

The LookForColourGoal() function is outlined in Figure 4.9.

The first thing to ask, again, is whether the whole CPlane is below the horizon. If it is, then the function can return immediately. This is particularly useful for filtering out little yellow patches on the red robot body.

The function will then initialise a dummy goal, which contains no bounding box or blobs information. So that from line 7 to line 20, all the blobs of the interested colour can be gone through and added to the dummy goal if they pass all the sanity checks.

In line 9, 10 and 11, simple checks are applied to the current blob in the loop. If the blob is already part of a recognised beacon, then it cannot be a goal. If its area is too small, or it is too high above or too low below the horizon, it will also be thrown away.

From line 14-16, information provided by recognised beacon is used to justify whether the blob can be a goal or not. Here the importance of beacon recognition is shown again — these checks compose a large proportion of goal sanity checks. These sanity checks are gone through in detail below according to their execution order. Note that the potential goal blob that the for- loop currently points to is called “the blob” in the sanity checks below, to save lengthy wordings.

1. Is the blob higher than the beacon? (line 14)

```
1   LookForColourGoal(Blue/Yellow, Left beacon, Right beacon) {
2
3       Is whole CPlane below horizon?
4
5       Initialise an empty goal
6
7       Go through each blue/yellow blob to try to add to the goal
8
9          Is blob already matched as a beacon?
10         Is blob size too small?
11         Is the blob too high or too low compare to horizon?
12
13         Go through each recognised beacon
14             Is the blob higher than the beacon?
15             Is beacon a lot higher than the blob?
16             Is the heading angle between the blob and beacon is too small?
17
18         Is the blob dimension too small?
19         Is the blob in the left of left beacon?
20         Is the blob in the right of right beacon?
21         Is the blob too far apart from current goal?
22
23     If a goal is formed now,
24
25         Is goal area too small?
26         Is goal density too low?
27         Is the width/height ratio far from normal?
28         Is there enough green pixels nearby?
29         Are the width and height in reasonable length?
30
31     Otherwise,
32         Return
33
34     If width/height is not right,
35         Apply correction
36
37     Assign all the details to the goal visual object.
38 }
```

Figure 4.9: Pseudo Codes - LookForColourGoal()

Since the goal is always lower than any beacon, this check is very useful in eliminating noises that come from the background.

2. **Is beacon a lot higher than the blob? (line 15)**
   On the other hand, a goal blob cannot be too low below a beacon either because they are all fixed. This is helpful in eliminating yellow reflections in the barrier of the soccer field or edges of red robots that

turned yellow occasionally.

3. **Is the heading angle between the blob and beacon is too small? (line 16)**

   Under the limited size of the soccer field, it is found that the horizontal angle between the goal and its neighboring beacons has a lower bound. A diagram in Figure 4.10. Therefore a yellow blob that is observed close to a beacon cannot possibly be a yellow goal. An exception is the viewpoint of a goalie, which maybe looking at the beacon on the side while seeing part of a goal. This exception is specially handled.

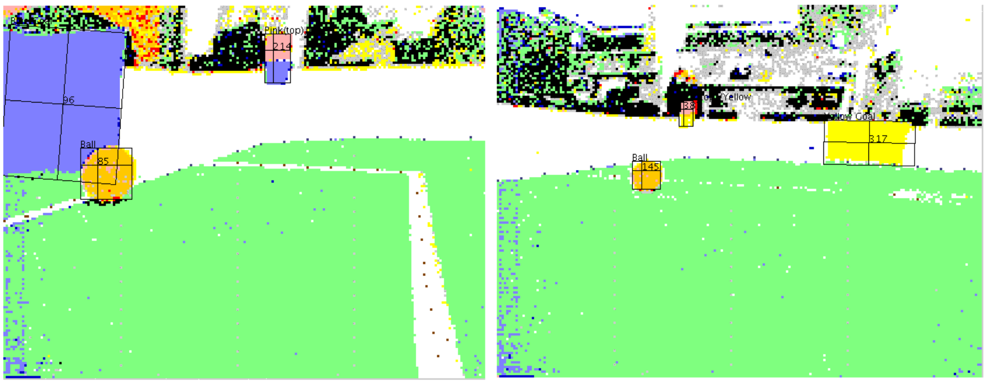


Figure 4.10: Constant Angle between Goal and Beacon from Different Viewpoint

From line 18-21, further checks are used to ensure the blob is part of a goal. If sanity checks are all passed, the blob will be added to the dummy goal object. These checks are explained below.

1. **Is the blob dimension too small? (line 18)**
   A standard check to eliminate background noise.

2. **Is the blob in the left of left beacon? (line 19)**
   This is the place where the information of input beacons is used. For example, if a blue blob is observed in the left hand side of the Blue Left beacon, it is impossible to be part of a goal.

3. **Is the blob in the right of right beacon? (line 20)**
   This is the opposite of the above check.

4. **Is the blob too far apart from current goal? (line 21)**
   This is the last check for the blob. If this is the first blob that come

across this check, it will be added to the dummy goal straight away to enlarge its bounding box. Otherwise, it will be compared with the information of the current dummy goal, to judge whether it is also part of the goal. Factors in consideration include how far this blob is from the current goal box, and also what colour of pixels are in between them.

In line 23, any blob that belongs to the goal would have been saved in the dummy goal object. If the dummy goal object is still empty at this point, no goal can be observed and the function can return straight away. If the dummy goal object is not empty, further sanity checks shown below are applied.

1. **Is goal area too small? (line 25)**
   Although there were already sanity checks about areas of blobs above, but they were using small threshold values because blobs from a scattered goal are expected to be small. Now that the goal is formed, size check will be applied again using greater threshold values.
2. **Is goal density too low? (line 26)**
   Density is ratio of area of blobs that belong to the goal divided by the area of the bounding box of the goal. If this value is too small, that means the blobs are too scattered and the goal will be ignored.
3. **Is the width/height ratio far from normal? (line 27)**
   This checks if the goal has an extreme shape. For example if a potential goal is 100 pixels wide but 2 pixels high, then this sanity check will filter it out because it is too extreme.
4. **Is there enough green pixels nearby? (line 28)**
   A real goal is always placed on the soccer field and therefore there should be some green pixels below the potential goal in the image. This property is used in this sanity check.
5. **Are the width and height in reasonable length? (line 29)**
   Finally the absolute width and height of the goal will also be checked separately. If it is too narrow, too wide, too tall or too flat, it will be ignored.

After these sanity checks, the goal is confirmed to be in place and details will be copied to the goal visual object. In line 34, the width/height ratio of the bounding box of the goal will be checked again, not as a sanity check but a condition to modify the height. This is needed because the height could be very inaccurate due to obstruction of other objects like ball or robots, and it is the only factor that determines the distance between the robot and the goal. Distance estimation will be discussed in detail in Section 4.4.3.

4.4.2 Logical Checks Among Recognised Goals and Beacons

After the recognition of beacon and goal objects, LandmarkSanities() function will then ask the following questions:

1. **Are yellow beacons and blue goal seen at the same time?**
   If any of the yellow beacons and the blue goal are recognised at the same time, one of them must be wrong. A simple decision is made in this case — throw away the goal. This is base on the fact that beacon is composed of two blobs and have many more sanity checks, thus the chance of getting false-positive beacons is lower than that of false-positive goals.

2. **Are blue beacons and yellow goal seen at the same time?**
   This is same as the above question.

3. **Are yellow and blue goals seen at the same time?**
   When goals of opposite colour are observed at the same time, one of them must be false. In this case, the variances of both goals are compared and one with lower variance will be kept.

   Variance is a value given to recognised object in the vision module, to indicate how certain the system think that the object is in the estimated location. The formula that calculates the variance for goal is given in (4.3), which is left unmodified from previous year of rUNSWift because this value is functioning well with the new model of ERS-7.

$$goalVariance = \frac{400}{goalHeight} \tag{4.3}$$

4.4.3 Goal Distance Estimation

The distance estimation used for goal is very similar to that for the beacons. In goal, however, there is no distinct feature like the centroid-to-centroid distance in beacons. Therefore the height of the bounding box of the goal is used instead to indicate the distance between the observing robot and the goal.

The formula that relates the distance and height is shown in (4.4), where c1 = 5818.6 and c2 = -0.2054. This formula was from rUNSWift 2003 and Derrick Whaite from rUNSWift 2004 has modified its constants in 2004 using experimental results with the new model of ERS-7.

$$goalDistance = \frac{c1}{height + c2} \tag{4.4}$$

4.5 Balls

The soccer ball is the drive of the game. It is very important to keep track of the ball so that the robot can position itself in the best position to kick it into the right direction. On the other hand it is just equally important, if not more, not to see any false ball because that will turn the behaviour of the dog upside down.

4.5.1 Ball Recognition

After the landmarks are recognised, ball recognition starts. In many ways it is simpler than goal and beacon recognition because we assume that for any ball it only has one blob shown in the CPlane. Therefore there is no need to check the neighbouring orange blobs and try to join them together.

The distance estimation of the ball, however, is much more complicated than those of beacon and goal. Depending on different sizes of the ball blob and orientations on the CPlane, different methods were used to estimate the distance between the observing camera and the ball. These distance estimation methods will not be discussed in this paper, however, because they have been discussed in details in the rUNSWift 2003 report (Section 2.3.4 Ball Recognition, p.37) in the Bibliography list, and these methods have largely remain unchanged in 2004 except for small modification of constants by Kim Cuong Pham during his porting work from ERS-210 to ERS-7 in the summer of 2003/2004.

The main problem in 2004 faced by ball recognition is to filter out the orange patches that appear on the edges of red robot uniforms. This is a combinatory effect due to the poorer vision of ERS-7, a more serious chromatic distortion and the unstable lighting condition.

Details of FindBall()

The FindBall() function is relatively simpler than the recognition of goals and beacons: Go through each orange blob, stop and return if any one in the list is found to be ball. An option for future development may be not to stop when the first potential ball is found, but go ahead to see if there are orange blobs that are more likely to be ball compare to the first one. An outline of FindBall() is shown in Figure 4.11.

```
FindBall() {

    Go through each orange blob

        Is this blob too flat or too thin?
        Is the density too low?
        Is any corner of the ball higher than the horizon?

        If this ball has a big area
             Calculate the details of the ball using "fireball" method.

         Calculate the details of the ball using normal method

         If the whole screen is covered
             OR ball passes checks in BallSanities(),

             Is the ball small but calculation gave small distance?
             Is the ball distance too large?
             Is ball too high up on ground?

             If ball has large area,
                 Remove beacons in edges

             Calculate further details of ball
             Return from function because a ball is found.

         Otherwise,
             This blob cannot be a ball,
             skip to next blob
}
```

Figure 4.11: Pseudo Codes - FindBall()

The function first checks the size and density of the blob (density is the blob area divided by bounding box area, just like those for beacons and goals), to eliminate small noises of orange pixels in the background or on the red robot. It then check if any corner of the ball is higher than the horizon, and throw it away if it does. Corners of the blob are checked instead of the centroid of the blob, because sometimes noises may happen to have their centroids close to the threshold values, but not all of their corners.

In line 9-12, the details of the potential ball including its distance are estimated in different ways depending on its size and orientation in the CPlane. The term “fireball” is used to describe large orange blob that happens when the ball is very close to the observing camera.

In line 14-15, a group of sanity checks are implemented in BallSanities() function, which will be looked at in detail below. If the ball blob can pass all these sanity checks, then more checks will be applied and eventually get this blob accepted as the blob. If this condition in line 14 failed, then it would be concluded that this blob is not a ball, and the next orange blob will be looked at.

Line 17-19 performed more checks after the condition above is satisfied. The reason why they are not grouped in BallSanities() function is because they require distance or other information that cannot be carried along by passing the potential blob. These checks will be looked at in more detail below:

1. **Is the ball small but calculation gave small distance? (line 17)**

   If this happens, this means something is wrong with the shape of the ball, which has led to a large distance while the ball is actually quite small in the CPlane. In this case the blob is no longer trusted and thrown away.

2. **Is the ball distance too large? (line 18)**
   This is a “desperate” sanity check added approaching the final stage of development. It was found that other sanity checks, even after fine-tuning, still could not eliminate some of the small orange blobs that appeared on the red robot uniform. Because this kind of orange blob often appears as a ball far away, a quick solution here is to ignore ball with great distance. The downside is that this threshold is a bit low, at about 3 to 3.5 metres, for it to be usefully eliminating noisy orange blobs.

3. **Is ball too high up on ground? (line 19)**
   The elevation angle and the distance estimation to the ball is combined here to estimate the height of the ball. If it appears to be too high up from the ground, this ball will be ignored.

After the above tests, in line 21, the ball is confirmed to be seen. A special check is added here to eliminate recognised beacons, if a beacon happens to be recognised in the edges of the CPlane and the ball is very large. This check is in place because it is found that orange pixels of the ball often turn yellow and pink in the corners and edges of the CPlane, as shown in Figure 4.12.

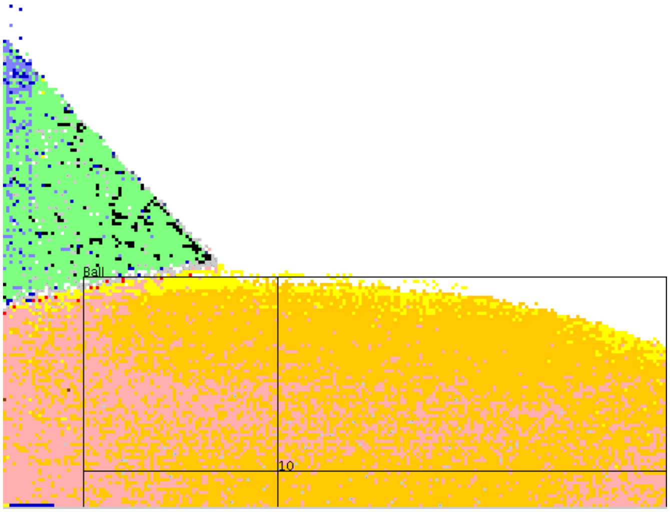


Figure 4.12: Close-up Ball that turns Pink andYellow

In line 25, all required checks are already made to the current blob and they are passed, the function will then return without checking the rest of the orange blobs.

Details of BallSanities()

BallSanities() function is called by FindBall() function to perform a list of sanity checks on the potential ball blob. An outline of this function is shown in Figure 4.13.

In line 3 and 4, the elevation of ball blob and that of goal and beacon are compared. Make use of our knowledge that ball can never be flying above ground; this check is handy and accurate most of the time.

The sanity checks in line 5 and 15 try to see if the ball is an effect of the chromatic distortion, although they are using different attributes. Sanity checks in line 6, 8 and 11 are all concerned with red colour, to see if the orange blob is actually part of red robot uniform.

```
1 BallSanities(Potential Ball blob) {
2
3     Is ball elevation higher than that of goal?
4     Is elevation higher than that of beacon?
5     Is ball on edges and scattered?
6     Are pixels around bounding box too red?
7
8     If area < 1000
9         Is the blob completely inside the largest red blob?
10
11     If area < 40
12         Is blob touching red blob?
13
14     If area < 150
15         Is the density of the ball too small?
16
17     If yellow goal is recognised inside the ball,
18         Reject goal
19 }
```

Figure 4.13: Pseudo Codes - BallSanities()

The last check in this function at line 17, try to eliminate recognised goal that is totally inside the ball. This is a good check, but it should actually be placed after line 20 in FindBall() function in Figure 4.11. Because that is the place when the ball is really confirmed to be seen.

4.6 Robots

Robot recognition is necessary in producing intelligent co-operation between robots. In particular, if teammate cannot be recognised by robot, they will run into each other easily and be caught for penalty.

4.6.1 Robot Recognition

Codes in robot recognition is considerable harder to read, because there are red and blue robots, and the recognition of them are not totally separated. The other complication comes from the fact that it is the only object with multiple blobs and multiple instances. This will be explained further in FindColorRobot().

A hard problem in robot recognition faced by rUNSWift 2004 is to recognise the blue robots. Recognising blue robots have always been more difficult than recognising red robots because solid blue is extremely hard to achieve in low-level vision. It is hard to achieve because the camera of the robot does not receive much light, and thus making the black and blue colour very similar. In 2004, however, the hard problem just got harder because of the poorer quality of camera.

Details of FindRobots()

The highest level of robot recognition starts with FindRobots() function. An outline of this function is shown in Figure 4.14.

First of all, RobotNoiseFilter() function is called to throw away those blobs with the robot red or robot blue colour as many as it can. Normal filtering conditions are implemented like the ones used in recognition of other objects that have been described, therefore RobotNoiseFilter() is not discussed in detail here. In line 5 the blobs are sorted from left to right, so that FindColorRobot() function can easily group blobs into robot.

In line 7 and line 13, blobs are further filtered out using the location of any recognised ball, before FindColorRobot() is called. Notice that red robots are recognised before blue robots. This is important because there are often blue or black colour pixels (which can easily turn to a blue dog) among red patches of the red robots with ERS-210 model. Therefore it is important to first recognise red robots present in the CPlane, then eliminate blue or black blobs around it.

In line 18, both blue and red dogs are already classified. More sanity checks are then applied and are discussed below:

1. **BLUE dog inside RED dog or vice versa? (line 20)**
   Although similar checks have been done above, but that was only applied to blobs when dog has not been recognised. Thus it is run again.
2. **Dog's height too large? (line 21)**
   It is found that the blue robots are sometimes merged with blobs in the background, which gave them large height and incorrect distances. It is preferred to miss a robot, rather than seeing a close robot because that could trigger a back-off action.
3. **BLUE dog in corner? (line 22)**
   Under the chromatic effect, the edges and especially corners of CPlanes often turns very dark, thus causing blue dogs to be recognised sometimes. A test is therefore implemented to prevent this.

```
1 FindRobots() {
2
3     RobotNoiseFitler() to apply general sanity checks
4
5     Sort the red/blue blobs from left to right
6
7     Go through each RED blobs
8         Is it near/below ball?
9
10     FindColorRobot(ROBOT_RED);
11
12     Go through each BLUE blobs
13         Is it near/below ball?
14         Is it near/inside red dog?
15
16     FindColorRobot(cROBOT_BLUE);
17
18     Go through each recognised DOG
19
20         BLUE dog inside RED dog or vice versa?
21         Dog's height too large?
22         BLUE dog in corner?
23         Dog very high above ball?
24         Is dog too far?
25         Dog has very large aspect ratio on bounding box?
26
27         Go through each beacon,
28             Dog's distance further than the beacon's distance?
29             Dog is above beacon?
30
31         Go through each goal,
32             Dog's distance further than the goal's distance?
33             Is the dog too high above goal?
34             If goal completely inside robot,
35                 Reset goal
36
37         Is recognised ball small and inside/next to red dog?
38             Remove ball
39 }
```

Figure 4.14: Pseudo Codes - FindRobots()

4. **Dog very high above ball? (line 23)**
   This test has been applied on individual blob, and is applied again on recognised robot.

5. **Is dog too far? (line 24)**
   Phantom robots that come from the background of the CPlane are

often very hard to eliminate. This test therefore sacrifices the ability to see far robots (about 3 metres) in exchange for removing phantom robots. This is not doing harm, however, because robots so far are not important for behaviour consideration in 2004 Behaviour module.

6. **Dog has very large aspect ratio on bounding box? (line 25)**
   If the recognised robot is too thin or too flat, it will be thrown away.

In line 28-34, more sanity checks are applied using information of recognised beacons and goals. In cases when the distance of the dog is impossibly greater than that of beacon or goal, the recognised robot will not be thrown away, but its variance will set to a very large value so that the localisation can use only the angle to the robots, but not their distances.

After line 34, robots are confirmed to be recognised. Therefore in line 36 and 38, goals and ball are thrown away if they are observed in an impossible way. An example of when this happens is shown in Figure 4.15. The danger is that there might be false robots that filter out real goals/ball! However, experimental results showed that the opposite happen most of the time.

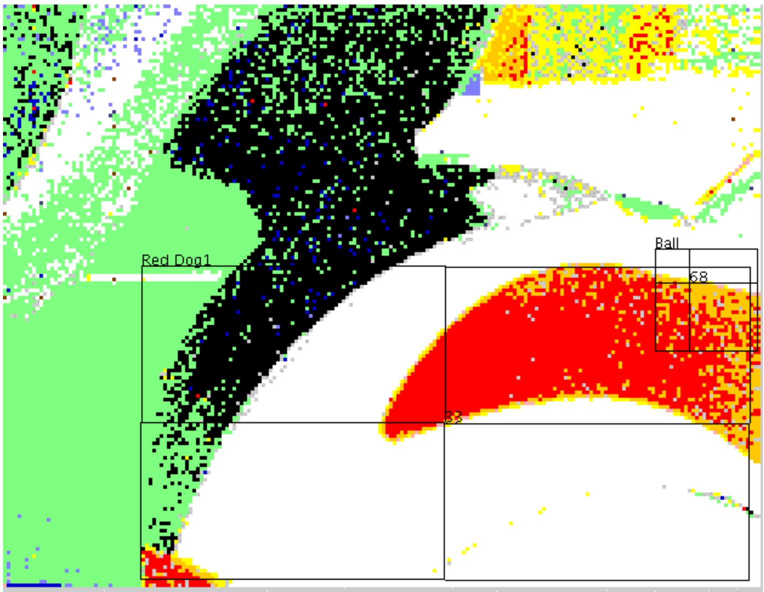


Figure 4.15: Red Robot Noise becoming Ball

Details of FindColorRobot()

In FindColorRobot() blobs of robots are considered and distributed into different robots. An outline of the function is shown in Figure 4.16.

```
1 FindColorRobot(blue/red) {
2
3     Initialise an empty robot
4
5     Go through blob of input colour from left to right
6
7         If this is the first blob
8             OR it is not far from the current robot
9
10             Add it to be part of current robot
11
12         Else
13
14             If 3 robots have been recognised
15                 Break out of the loop
16
17             Is last dog's bounding box very tiny? Y-> cancel it.
18
19             Start a new robot
20
21     Go through each recognised robot
22         Filter out blue robot with too much "MaybeBlue" colour
23         Calculate the details of robots
24 }
```

Figure 4.16: Pseudo Codes - FindColorRobot()

In line 3 an "empty robot" is initialised. This is like initialising an empty set, then put blobs that belong to the first robot into it. Line 7 is doing exactly that. Whether a new blob is to be put in the current robot depends on their horizontal and vertical distances. In this loop, a limit on number of observable robots is given in line 14. Also, sometimes a new robot is started but it is then found out to be too small (having just one small blob, for example), in that case line 17 will cancel it.

After the loop, all recognisable robots will be stored, and details of robots like distance will be calculated. However, just before that, a special sanity check on "MaybeBlue" colour for the blue robot is applied. This is used to help distinguishing the "real blue" colour – the colour of blue robot uniforms - from the black regions of the ERS-210 body. A special colour called "MaybeBlue" is labelled on pixels that the low-level vision module cannot tell for sure whether they are blue or black. So now in line 22, the ratio between the area of “MaybeBlue” pixels to the overall area can be checked and see if this recognised blue dog should be thrown away.

4.6.2 Robot Distance Estimation

Robot distance estimation was last modified in 2002, in which a linear relationship between the y-coordinate of lower edge of bounding box in CPlane and the robot distance is used to predict robot distance given the y-coordinate. The constants in this relationship were produced by running linear regression line over experiment data. This was largely unmodified in 2003.

In 2004, a different method is used and is outlined in 4.17. Instead of running linear regression line, a mathematical transformation developed by Kim Cuong Pham during summer of 2003/2004 was used to project the lower edge of the robot bounding box onto the ground to estimate how far the robot is from the camera. This transformation is implemented in getPointProjection() function and have been described in detail in the Kim’s 2003/2004 summer report.

```
Project the bottom line of the robot bounding box to the ground.

If the projected distance < 2 metres
    Keep this distance
Else
    Distance from height.
```

Figure 4.17: Pseudo Codes - Determine Robot Distance

If the distance is more than 2 metres and it is likely that the paw patches of the ERS-7 robot are too small to be included in the bounding box. Therefore the height of the robot bounding box is used to predict the distance instead, as shown in (4.5), where c1 = 1517.

$$RobotDistance = \frac{c1}{RobotBoundingBoxHeight} \tag{4.5}$$

This additional formula was produced by Kim Cuong Pham, in around June nearing the end of development in 2004.

4.6.3 Heading Estimation

In 2002, headings of robots were estimated using statistical heuristic methods base on the blobs distribution of the robot bounding box. This helps the behaviour module to produce fine-tuned actions that take the behaviours of opponents into consideration. This was largely unmodified in 2003.

In 2004, this task has not been tackled because both object recognition and behaviour modules have other work that kept the team busy.

4.7 Debugging Methods

For debugging work in vision module, it has the luxury of saving CPlanes in game condition, then revise the details of the program by compiling and running source codes offline using OffVision, or browse the CPlanes like watching a movie and see the fired sanity errors offline using CPlaneDisplay. In this section these tools will be introduced below, together with some special ways of debugging used in object recognition.

4.7.1 Display Devices

CPlaneDisplay

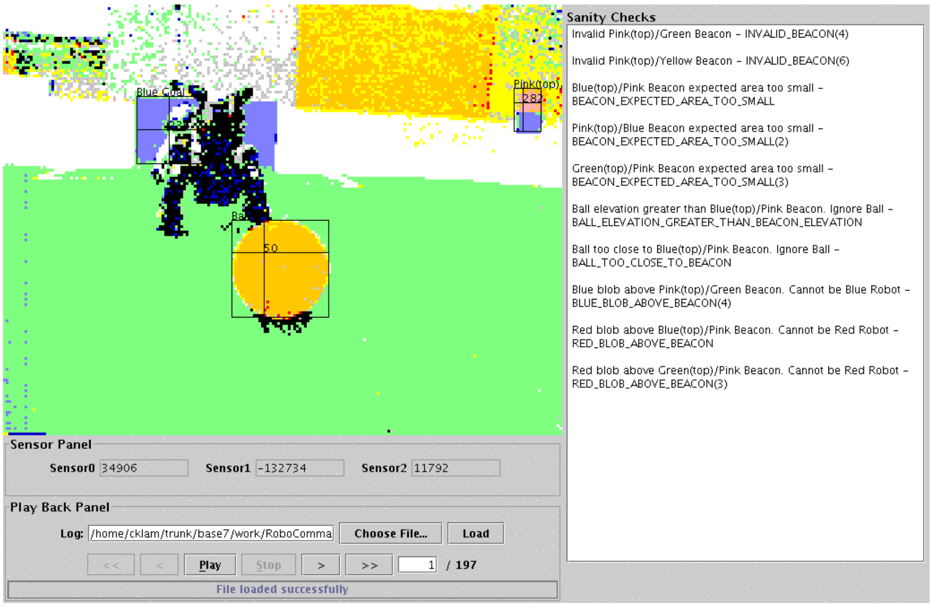
CPlaneDisplay is a java program developed by rUNSWift in 2002, which can be used to display logs of CPlanes online or offline. A screen shot is shown in Figure 4.18.

When the program is running offline, logs of CPlanes saved in RoboWirelessBase can be displayed and browsed frame by frame. Because sanity errors that are fired by sanity checks are also saved in the CPlanes, corresponding sanity error message that are fired by the object recognition module while the system was running can be re-displayed for each frame — offline.

The downside of using CPlaneDisplay as an offline debugging tool is that the sanity error messages are often not detail enough to track down the problem — it can only indicate what type of problems have occurred, but it will not tell the user which blob has caused the problem, or other information of the blob.

OffVision

Because of the downside of the CPlaneDisplay, a more powerful tool was developed by rUNSWift in 2002 as well to run the entire vision module of-



Sanity Checks

```
Invalid Pink(top)/Green Beacon - INVALID_BEACON(4)

Invalid Pink(top)/Yellow Beacon - INVALID_BEACON(6)

Blue(top)/Pink Beacon expected area too small -
BEACON_EXPECTED_AREA_TOO_SMALL

Pink(top)/Blue Beacon expected area too small -
BEACON_EXPECTED_AREA_TOO_SMALL(2)

Green(top)/Pink Beacon expected area too small -
BEACON_EXPECTED_AREA_TOO_SMALL(3)

Ball elevation greater than Blue(top)/Pink Beacon. Ignore Ball -
BALL_ELEVATION_GREATER_THAN_BEACON_ELEVATION

Ball too close to Blue(top)/Pink Beacon. Ignore Ball -
BALL_TOO_CLOSE_TO_BEACON

Blue blob above Pink(top)/Green Beacon. Cannot be Blue Robot -
BLUE_BLOB_ABOVE_BEACON(4)

Red blob above Blue(top)/Pink Beacon. Cannot be Red Robot -
RED_BLOB_ABOVE_BEACON

Red blob above Green(top)/Pink Beacon. Cannot be Red Robot -
RED_BLOB_ABOVE_BEACON(3)
```

Sensor Panel

Sensor0 34906 Sensor1 -132734 Sensor2 11792

Play Back Panel

Log: /home/cklam/trunk/base7/work/RoboComma

1 / 197

File loaded successfully

Figure 4.18: A Screenshot of CPlaneDisplay

fline. This tool therefore allows full debugging in object recognition, as all the blobs information will be available offline as well. A snap shot of this tool together with the debugging messages are shown in Figure 4.19 and Figure 4.20.

For even easier debugging, new features added to this tool in 2004 include grids and horizon line drawn on CPlane, radio box in the bottom that allows showing of a particular colour, and more systematic debugging messages.

CPlaneClient

CPlaneClient is a display tool developed by Kim Cuong Pham in 2004. It is only for online display, but it has its advantage over the older CPlaneDisplay that it can display CPlanes in higher speed. A screen shot is shown in Figure 4.21.

4.7.2 Debugging Techniques

Offline Debugging Messages

The offline debugging messages in 2003 were largely arbitrary and messages are added to the OffVision output anytime in anywhere of the code, in any

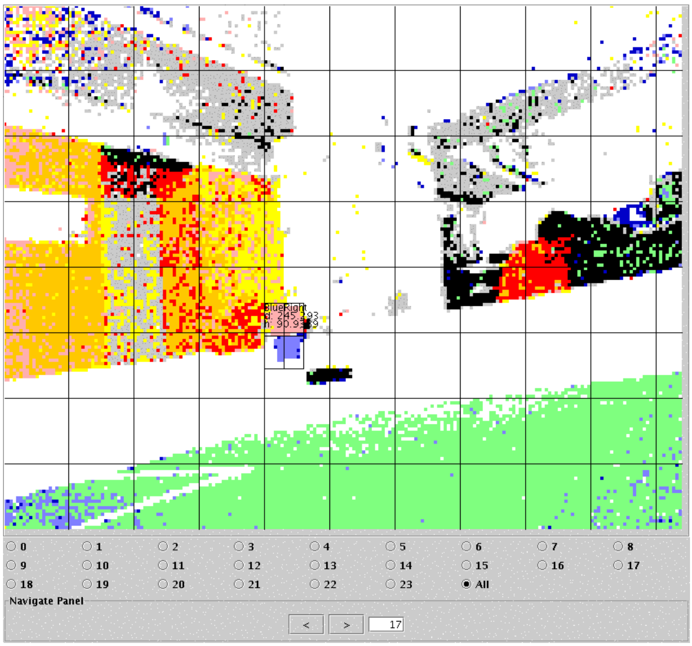
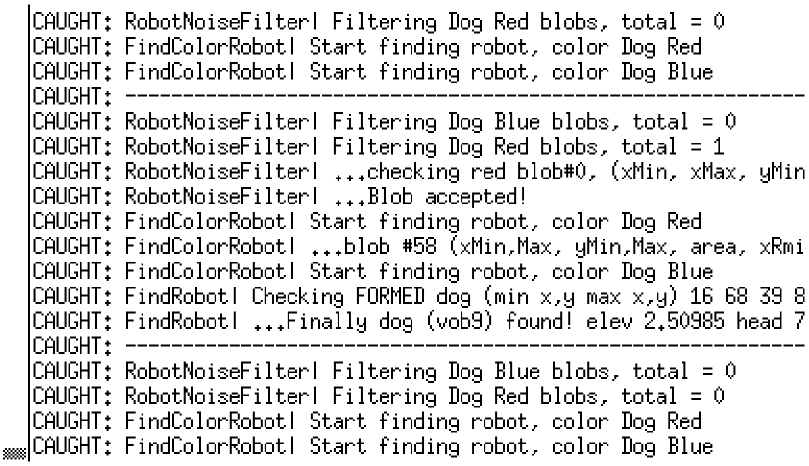


Figure 4.19: A Screenshot of OffVision

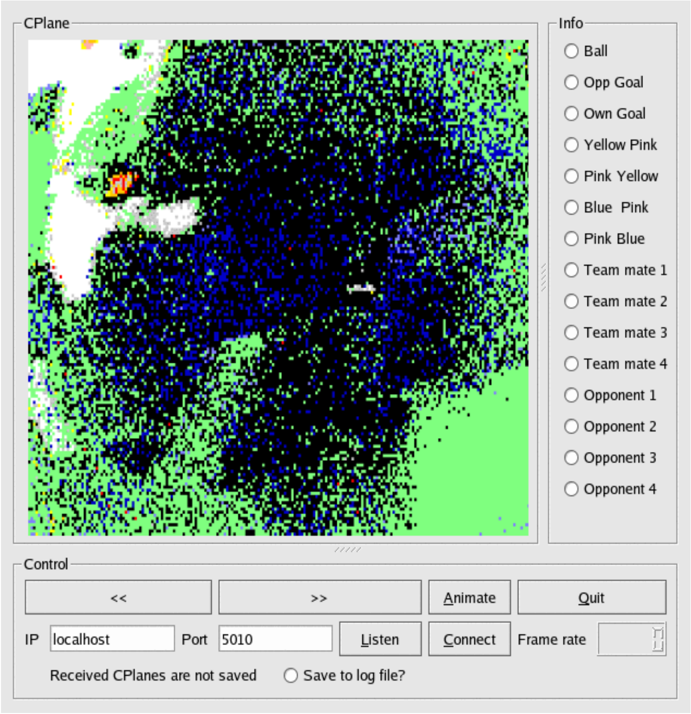


```
CAUGHT: RobotNoiseFilter| Filtering Dog Red blobs, total = 0
CAUGHT: FindColorRobot| Start finding robot, color Dog Red
CAUGHT: FindColorRobot| Start finding robot, color Dog Blue
CAUGHT: ----------------------------------------------------------
CAUGHT: RobotNoiseFilter| Filtering Dog Blue blobs, total = 0
CAUGHT: RobotNoiseFilter| Filtering Dog Red blobs, total = 1
CAUGHT: RobotNoiseFilter| ...checking red blob#0, (xMin, xMax, yMin
CAUGHT: RobotNoiseFilter| ...Blob accepted!
CAUGHT: FindColorRobot| Start finding robot, color Dog Red
CAUGHT: FindColorRobot| ...blob #58 (xMin,Max, yMin,Max, area, xRmi
CAUGHT: FindColorRobot| Start finding robot, color Dog Blue
CAUGHT: FindRobot| Checking FORMED dog (min x,y max x,y) 16 68 39 8
CAUGHT: FindRobot| ...Finally dog (vob9) found! elev 2.50985 head 7
CAUGHT: ----------------------------------------------------------
CAUGHT: RobotNoiseFilter| Filtering Dog Blue blobs, total = 0
CAUGHT: RobotNoiseFilter| Filtering Dog Red blobs, total = 0
CAUGHT: FindColorRobot| Start finding robot, color Dog Red
CAUGHT: FindColorRobot| Start finding robot, color Dog Blue
```

Figure 4.20: Debugging Messages from OffVision

form preferred by the programmer at the time. This made the messages written in 2003 completely unreadable in 2004.

In 2004, variables are created for each function to be debugged, like "debugFindBall", "debugFindBeacon", etc. These variables act as switches



IP localhost Port 5010

Received CPlanes are not saved

Figure 4.21: A Screenshot of CPlaneClient

to turn on or off offline debugging messages of their corresponding function. Also, all offline debug messages are printed with prefixes like "FindBall— ", "FindBeacon— ", so that messages can easily be traced back to where they were fired.

Send CPlane with Conditions

Browsing and reviewing logs of CPlanes have been very useful debugging activities — but sometimes a particular problem is interested only in a small proportion of the CPlanes, then the browsing exercise in the thousands of logs would become very boring.

Because of this, slight modifications were made on the codes running in the robot so that not all the CPlanes will be sent to the RoboWirelessBase. Instead, they are only sent when certain conditions are met. For example, it can be specified to send CPlanes only when a ball is not recognised. With this setting, and let the robot play with a soccer ball, it is then easy to check if there is any ball unrecognised when we look at the CPlane logs sent by the robot. It will not contain a lot of frames, and any ball showing on those frames would be the unrecognised missing balls.

Chapter 5

Problems Encountered and Possible Solutions

Throughout the object recognition development, there were some particular problems that significantly affected our overall progress. In this chapter we will discuss these particular stumbling blocks, together with our attempted solutions and possible solutions that are still on paper.

5.1 Unstable Lighting Condition

In 2003, the lighting during a game condition is provided by four floodlights and two studio lights. In the original rules of 2004 Four-Legged League competition, it was suggested that the two studio lights should be removed to make the lighting condition closer to natural settings. After some period of experiments with the new model of ERS-7 and the new setting, however, this change was cancelled because it poses challenges too big to the vision module.

During the time that the rule has not been reversed back, the development of object recognition was slowed down because of attempts to cope with the different lighting condition, while still struggling with other problems of ERS-7 listed below.

5.2 Chromatic Distortion

Also known as the Ring Effect, the chromatic distortion of the colour camera refers to how the colour of the pixels that lie near the edges of the image appear differently from the pixels at the centre, while they are actually having the same real colour. An example is shown in Figure 5.1.

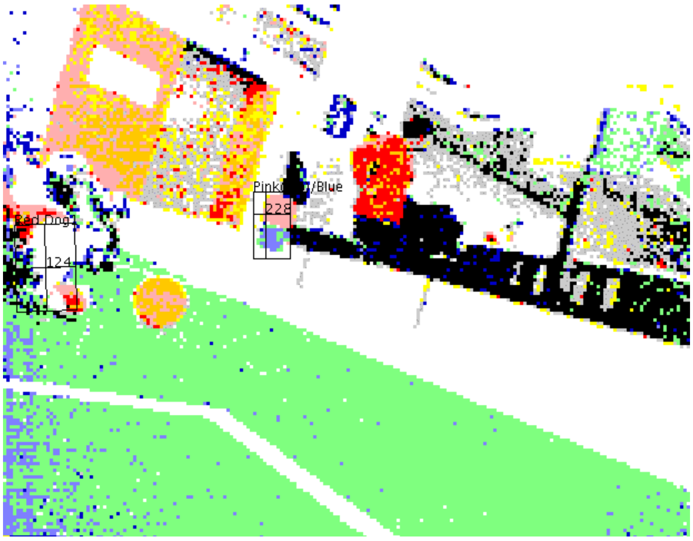


Figure 5.1: The Ring Effect (blue noises along the edge)

This chromatic distortion also existed in older models. However, this problem becomes far worse in the new model of ERS-7 comparing to the old model of ERS-210. With the old-model, the effect is confined to some noise of few pixels in the corner of an image, while for the new model a ”ring” with a width of more than 10 pixels can be seen in most samples. Under such effect yellow blobs in beacons often turn orange in edges, and red blobs in robots turn yellow or orange. This has created a great burden for higher vision object recognition, as the balancing between accepting more false-objects classifications or missing real objects become much harder to achieve. This in turn made the behaviour of the robots more difficult to develop as the sensing of the environment is less reliable.

A lot of effort was put into eliminating this undesirable property of the camera from the beginning of the development. In around April, after the Australian Open of RoboCup 2004, Jing Jing Xu from our team developed a technique that can counteract the distortion - what we called ”The Ring Offset”. It was not perfect, but this technique already greatly reduced the effect compared to the original distortion.

5.3 Different Robots, Different Cameras

Beside the serious chromatic distortion, it is also found that different cameras in different robots produce quite different images in terms of their raw values. Therefore, although two robots that are using the same colour calibration file are looking at the same object under the same lighting condition, the CPlanes produced by these robots are still different. These differences are subtle among some robots but in some cases the difference is serious enough to misclassify an object. For example, when a particular robot see a lot of yellow in the white barriers while other robots would only see white, this particular robot can then easily miss out yellow beacons and yellow goal because the yellow blobs of these objects may merge with the yellow noises in the white barrier.

This problem has created confusion when attempts were made to reproduce errors in object recognition – misclassification occurred when tested in robot A but could not be reproduced in robot B. The cause of this problem was discovered early in our development. However, because of other important issues on our agenda, this problem was not tackled until a later stage.

To ensure different robots will perceive the same optimised colour under the same lighting condition in spite of this undesirable camera problem, we can either

1. Create a colour calibration file for the required lighting condition for each robot – i.e. take a full set of calibration photos using cameras of every robot, or
2. For each robot, modify the way the default calibration file is used, so that even though each robot has slightly different raw values in their images, the produced classification is still the same.

The first option can achieve an ideal quality of colour classification for each robot, but is infeasible in our case because creating a colour calibration file takes a long time – usually more than 3 hours – while we only have a very limited preparation time during the competition. Therefore the second approach is used and in May 2004, a method was invented and applied to the vision system by calculating and applying what we called "camera offset". This will be explained in the following section.

5.3.1 Calculating and Applying Camera Offset

Because we know that the same colour under the same lighting condition would be classified differently in different robots, there must be some deviation in the perception of YUV values of these robots to these colours. Thus our mission here is to find a transformation that will change YUV values of different dogs to be the same YUV values as the dog that is used for colour calibration. This experiment thus goes like this:

1. The dog that make the calibration and each dog that requires correcting will take photos of 8 paper with different colours. These 8 colours can be arbitrary, as long as they cover enough proportion of the colour spectrum. If we have 5 dogs (1 of them is the original photo taker for the calibration), then 5x8 = 40 photos need to be taken.
2. Then, Y, U and V values are separated and a formula will be produced for each component
3. To find the correction for U component for dog A, for example, we will try to look for the m and c variable in U[dogCalibration] = m * U[dogA] + c.
4. To find these variables, a table of average U values of different colour for dogCalibration and dogA will be recorded, and regression run on them. An example of such table is shown in Table 5.1. The resultant gradient and offset of this regression are our desired m and c required above.

	Dog A	Dog Calibration
Black	18	19
Dark blue	18	20
Dark red	19	22
Green	24	25
Yellow	86	91
Orange	62	68
Pink	42	42
Light blue	30	33

Table 5.1: An Example of Average Y values obtained by Dog A and Dog “Calibration”

After getting formulae like U[dogCalibration] = m * U[dogA] + c for every robot that require adjustment, the remaining task is apply them automatically. This can be done using MAC Address detection, because each robot is equipped with a unique MAC Address. The constants for different robots will be stored according to their MAC Address in the system, so that no matter which robot is loaded, the correct variables would be used.

Applying the offset does not solve all the problems, but it does have an effect of averaging the quality of the cameras.

5.4 Error-Prone Manual Adjustments

With problems illustrated in sections above, in addition to developments and revision of sanity checks, the threshold values of sanity checks need to be updated from time to time. The amount of change in threshold values that should be applied each time is often hard to determine and entered by intuition. This makes questions like, "Where did this value come from", or "How do you know this amount is correct", often asked during the development cycle, very hard to answer with convincing empirical results.

Until 2004, this mode of development is still used by rUNSWift and it will continue unless we can come up with a more organised and systematic way of modifying threshold values. One such method is suggested - Rule-Base Learning.

5.4.1 Rule-Base Learning

The aim of rule-base learning is to accumulate a knowledge base so that the values of thresholds in sanity checks can be explained by tracing back relevant examples. It can also show the amount of true/false positives and true/false negatives using the samples stored in its database. A simple algorithm that has yet to be implemented is provided here (fig. 5.2).

In fig. 5.2, object T is one of the objects to be recognised, i.e. beacon, goal, robot or ball. KB stands for Knowledge Base, representing the database that stores sanity check rules and sample images. Note that, for illustration of rule-based learning, the outlined algorithm is simplified to recognise only objects composed of one blob. To apply this algorithm to recognition of objects with multiple blobs like beacons and robots, the relationship between current blob and surrounding blobs need to be considered as well.

We may not always add a rule for false-positive because there may be cases that are not so clear-cut – Sometimes part of the robot really look like a ball, and the ball may appear like a robot. It could be caused by

```
Given an image with blobs information

For each blob B in the image,
    For each type of object T,

        Consult the current sanity check rules of that object
        If no rule is violated,
            Classify blob B as object T

        Consult human judgment,
        If auto-classification for this blob is false
            OR this blob is auto-classified as two objects,

            Design a new rule to eliminate false classifications

            If the new rule does not contradict many old classifications,
                Save this image, AND
                Add a rule to the KB to make this sample a true-negative

            Otherwise,
                Cancel the rule, AND
                Leave this example as a false-positive

Repeat again for any image
```

Figure 5.2: Sample Algorithm of Rule-Base Learning

problems in lighting or low-level vision processing. Therefore, the system should check any new rule with the old samples in its knowledge base.

This is an idea discussed after finishing RoboCup 2004. The difficult part of the implementation will likely to be the linking and merging between the samples database, sanity checks in the robot software and the program itself that generate the sanity checks.

Chapter 6

Conclusion

This paper described the development of an object recognition system that was used to recognise beacons, goals, balls and robots in the Four-Legged League of RoboCup Competition environment.

Currently, the system is far from being “autonomous” in the sense that it cannot automatically cope with changes in lighting condition, specifications of objects, or any assumption implied by the gaming condition. The system works because all required knowledge about the objects is already hard-coded as rules, entered by human hands before the system runs. This research is yet another example that shows how a task that can be performed by human beings easily is in fact so complicated.

Nevertheless, the system worked reasonably well in a game environment, and can cope with changes in lighting condition to some extent without modifying the codes. To achieve more ambitious “meta-goals” like shortening development time, automatic learning of rules and minimising human intuitive intervention, however, more research and development effort will be required.

Bibliography

[1] Chen J., Chung E., Edwards R., Wong N., Hengst B., Sammut C., Uther W., *Rise of the AIBOs III - AIBO Revolutions*, School of Computer Science and Engineering, University of New South Wales, And National ICT Australia, 2003.

[2] Olave A., Wang D., Wong J., Tam T., Leung B., Kim M. S., Brooks J., Chang A., Huben N.V., Sammut C., Hengst B., *The UNSW RoboCup 2002 Legged League Team*, School of Computer Science and Engineering, University of New South Wales, 2002.

[3] Mak E., *Real-Time Obstacle Avoidance and Path Planning for Mobile Robot with Limited Onboard Processing Power*, School of Computer Science and Engineering, University of New South Wales, And National ICT Australia, 2003.

[4] Tang A., *Software Engineering and Experiments using Sony Four-Legged Robots*, School of Computer Science and Engineering, University of New South Wales, And National ICT Australia, 2003.

[5] Sheh R., *Visual Feature Detection for Robotic Soccer*, School of Electronic and Communication, Curtin University, 2003.

[6] Russell S., Norvig P., *Artificial Intelligence, A Modern Approach*, 2nd Edition, Prentice Hall, 2003.

[7] Murphy R., *An Introduction to AI Robotics (Intelligent Robotics and Autonomous Agents)*, Hardcover Edition, The MIT Press, 2000.

[8] Tom Mitchell, *Machine Learning*, McGraw Hill, 1997.

[9] RoboCup Technical Committee 2004 *Sony Four Legged Robot Football League Rule Book*, RoboCup

[10] RoboCup, *The RoboCup Official Website*, http://www.robocup.org

[11] Sony Coporation, *OPEN-R SDK Programmer's Guide 2004*, http://www.openr.aibo.com

[12] Sony Coporation, *OPEN-R SDK Programmer's Guide 2004*, http://www.openr.aibo.com

[13] Sony Coporation, *OPEN-R SDK Model Information for ERS-7*, http://www.openr.aibo.com

[14] Wikipedia.org *Color Space*, http://www.fact-index.com/c/co/color space.html