UNIVERSITÄT DORTMUND
■ FACHBEREICH INFORMATIK

Sony Legged League

Virtual Robot: Automatic Analysis of Situations and Management of Resources in a Team of Soccer Robots.

PG 442

Students: Damien Deom, Jörn Hamerla, Mathias Hülsbusch, Jochen Kerdels, Thomas Kindler, Hyung-Won Koh, Tim Lohmann, Manuel Neubach, Claudius Rink, Andreas Rossbacher, Frank Roßdeutscher, Bernd Schmidt, Carsten Schumann, Pascal Serwe

Supervisors: Ingo Dahm, Matthias Hebbel, Walter Nistico, Christoph Richter, Dr. Jens Ziegler

September 2004

FINAL REPORT

Lehrstuhl für Systemanalyse
Fachbereich Informatik der
Universität Dortmund

Computer Engineering Institute
Fachbereich Elektrotechnik der
Universität Dortmund

Figure 1: An early robot dog ..[1]

Figure 2: .. and the technology behind it.

[1] The picture shows Muffit, a character from the Battlestar Galactica TV series. NBC, 1978-80.

Contents

1 Introduction **4**

1.1 Overview of the RoboCup 4

1.2 Overview of the project group 5

2 Basics **6**

2.1 Rules of the games 6

2.2 API and operating system 6

2.2.1 Aperios 7

2.2.2 Open-R 7

2.2.3 GT2004 8

2.3 GermanTeam software architecture 8

2.3.1 Process framework 8

2.3.2 Module concept 9

2.4 RobotControl 10

2.5 Main focus of the GermanTeam 11

3 Tuning for the ERS-7 **12**

3.1 New hardware 13

3.2 New SDK 14

3.3 Software changes 14

3.3.1 Development of a new walking gait 15

3.3.2 New kicks and MOFs 18

4 Image Processing **27**

4.1 Motivation 27

4.1.1 Color Correction 28

4.1.2 Supporting Color Tables 28

4.2 EdgeDetection 28

4.3 Raster Image Processor 31

4.3.1 Architecture 31

4.3.2 Clustering 33
4.3.3 Ball Detection 34
4.3.4 Beacon Detection 36
4.3.5 Goal Detection 37
4.3.6 Line detection 38
4.3.7 Obstacle detection 44
4.3.8 Opponent Detection 44
4.3.9 Other Approaches to Opponent Detection 49
4.4 Image Processing for the Open Challenge 53
4.4.1 Climbing the ramp 53
4.4.2 Platform Beacon Detection 53

5 Resource Scheduling **56**
5.1 Dynamic Team Tactics 56
5.1.1 Overview on Dynamic Team Tactics 57
5.1.2 Files, Folders and Implementation 60
5.1.3 Working with DTT 61
5.1.4 The inner workings of option ratings 64
5.2 Scheduler Module Integration 66
5.2.1 Senso - a sample application 67
5.2.2 Conclusion 68

6 Ceiling Camera **69**
6.1 Lens distortion correction 70
6.1.1 Reduced distortion models 72
6.2 Perspective correction 72
6.3 Implementation 74
6.4 User interface 74
6.5 Towards an automated oracle 75

7 Competitions **77**
7.1 German Open 2004 77
7.2 Opens 2004 78
7.3 RoboCup 2004 78
7.3.1 WE ARE THE CHAMPIONS!! 79
7.3.2 Open Challenge 83

8 Side Projects **85**

8.1 World State Player 85
8.2 Demo Stick 86
8.3 Walking on a Leash 86

References **87**

Chapter 1

Introduction

This final report describes the work and the results of one year's work of the project group 442. The main topic and focus area of research was the further development of artificial intelligence concepts, cooperative decision making and collaborative solution development by autonomous robots. The virtual robot metaphor serves as a means of realization for all these tasks. Additional supportive fields of development were enhanced image processing to allocate additional computing time for decision making processes and the introduction of an overhead ceiling camera, which will be used for automated debug purposes and serves as an additional external control instance to enhance the visualization of robot internal data matched with real on-pitch situations.

1.1 Overview of the RoboCup

RoboCup is an international initiative which promotes the research and development of Artificial Intelligence and Robotics. The main focus of this project is in playing competitive soccer with robots, which means to examine and integrate technologies of autonomous agents, multi agent collaboration, real time planning and control and sensor data analysis and fusion. For this purpose the first official international conference and soccer games were held in Nagoya, Japan, in 1997. Followed by Paris, Stockholm, Melbourne, Seattle, Fukuoka and Padova, this year the 8th edition of the Robot World Cup Soccer Games and Conferences took place in Lisbon, Portugal.

Currently the RoboCup is also expanding in two other domains: the RoboCup Rescue and the RoboCup Junior League. Like the original RoboCup soccer, these are also divided into several leagues.

The games are important opportunities for researchers and developers to exchange technical information for advancing their own software and hardware solutions. So there is a large development from year to year. To keep this development interesting, a specific long-term objective was set. The RoboCup Federation set the ultimate goal for the challenge as follows:

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

For more information we reference to the official site from the RoboCup federation.[1]

1.2 Overview of the project group

The project group 442 (in the following just called “project group”) was based on the development and improvement of soccer playing robots. The robots that were used are of the type “Aibo” of the japanese manufacturer Sony.

To enable a reasonable kind of soccer game, it is necessary that several fields of robotic science like artificial intelligence, running movements, image processing and communication are combined and integrated cleverly. Therefore the soccer playing robots are of general interest for science.

This project group is part of the GermanTeam, which is composed of undergraduate students, PhD students and professors of the HU Berlin, the TU Bremen, the TU Darmstadt and the University of Dortmund.

The commonly developed code “GT2003” of the GermanTeam was supposed to be taken as a base for “GT2004” and further improved.

Because of a modular software-concept, it is possible to develop parts of a program together or in competition with other universities.

[1]RoboCup official Site: http://www.robocup.org/

Chapter 2

Basics

2.1 Rules of the games

The match is placed on a 4,60m x 3,10m large field and it is played with an orange colored ball (see picture 2.1). The goals are 60 cm wide and colored (yellow and sky-blue).

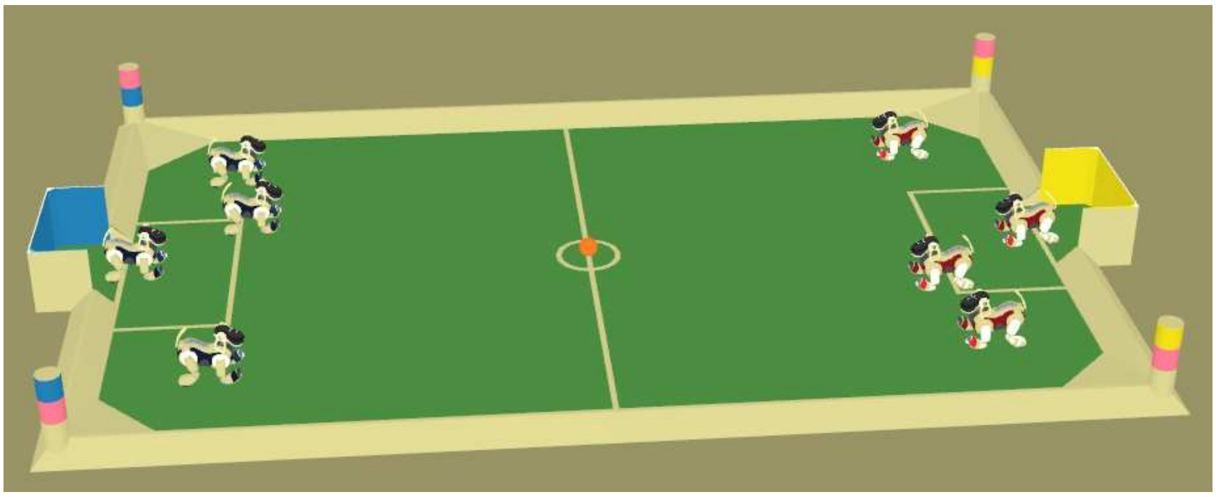


Figure 2.1: The Playing Field

Four robots form a team which can be identified by either the red or the blue colored tricots. At the corners of the field there are landmarks that help the robots to localize. The robots determine their position on the field looking at the goals and the colored landmarks. Every landmark has a unique color-code which is composed of the colors white, pink and either yellow or sky-blue, depending on the whether they are on the blue or yellow goal´s side. A match consists of two halves, each lasting 10 minutes. During the half-time interval the tricots and the sides will be switched. Further detail can be found in the official rules of the technical committee of the RoboCup.

2.2 API and operating system

In order to program the Aibo, Sony offers a development environment. This consists of the operating system Aperios and on top of it a Middleware API-library called "Open-R".

2.2.1 Aperios

Aperios is an operating system that was developed from the operating system Apertos and is applied in many consumer devices of Sony. The main characteristics are real-time capabilities and its object-oriented structure.

In Aperios each process is an object. Aperios enables the communication between two processes by message passing. This is information that is sent from a transmitting object to a receiving object. MESSAGES consist of a MESSAGEINFO-structure that contains information about type and size of the MESSAGE and the adequate data (i.e. camera-data).

Essential is the division of the objects into SENDER and OBSERVER. Each object must have the following functions (also called "Entry-Points" in the following):

- construction:
  - *DoInit ()*: initialization of an object
  - *DoStart ()*: start sending/ observing MESSAGES
- destruction:
  - *DoStop ()*: stop sending/ observing of MESSAGES
  - *DoDestroy ()*: removing the object
- subject-specific (Sender):
  - *ControlHandler ()*: establishing a connection
  - *ReadyHandler ()*: observing of MESSAGES
- observer-specific (Observer):
  - *ConnectHandler ()*: establishing a connection
  - *NotifyHandler ()*: observing of MESSAGES

2.2.2 Open-R

Aperios is not an operating system dedicated just to robots, on top of it there is another interface that provides the functions of the common programming of robots. This so called Open-R middleware API enables the access for all sensors (camera, sensing devices and so on) and actors (joints, LEDs, etc.) of a robot.

Open-R is an abstract API for all kinds of robots, i.e theoretically making possible to let the behavior of a four-legged robot run on a robot with wheels.

The fundamental components of the robot like joints, the camera, the LEDs and so on are called PRIMITIVES.

To use a sensor, the corresponding PRIMITIVE must be opened. During the initializing of a process the access is activated on a sensor by OPENPRIMITIVE. After that there is the possibility by CONTROLPRIMITIVE to change the settings (i.e. camera white balance adjustment). Then the data will be sent by MESSAGE and can be received by GETINFO and GETDATA and be analysed.

Another important feature is that the Open-R environment can also run on a personal computer which is very comfortable for testing a robot-simulation.

2.2.3 GT2004

GT2004 is the name of the complete project on which the project group has worked on. Since there are four universities taking part in it, there is a central CVS-Server (Concurrent Versions System, a file version control system which allows different persons to work on the same source code files) in Berlin, on which all source files are saved.

The structure of the complete project is laid out as follows. Each university has the option to develop their own ideas for sections of the project by themselves, to save these seperately and because of the modular structure of GT2004 they can be tested against each other.

A further aim of the modularization of GT2004 is to create an environment, where it is possibile to test the code on a robot and also on a simulator running on a Windows pc.

2.3 GermanTeam software architecture

Since the GermanTeam consists of several teams on different geographical locations, a software architecture which supports a cooperative, concurrent and parallel development is needed. To accomplish this goal the source code to control the entire robot is divided into encapsulated modules with well-defined tasks and interfaces and a process-layout where every process running on the robot is responsible for executing a set of modules.

2.3.1 Process framework

Processes in GT2004 are represented by classes which implement the system-independent interface PROCESS given by the Open-R framework. For Microsoft Windows the instances are realized as threads inside RobotControl(see 2.2.3), on the robot as Aperios processes. The main routine of this class is *main()* and has as return value the time in milliseconds until the routine starts again after finishing (if the value is positive) or parallel to the running routine (if the value is negative). Processes can communicate among each other through Message-Objects(see 2.2.1 on the previous page) The set of processes running concurrently on a system is summarized in a so called “Process-Layout”. In the current GT2004 Process-Layout three processes are running parallelly:

1. THE COGNITION PROCESS is responsible for the Image Processing(representated by the IMAGEPROCESS module [2.3.2]), the Behavior Control(BEHAVIORCONTROL module [2.3.2]), and the Worldmodel Generation(LOCATOR modules in Figure 2.2 on the following page).

2. THE MOTION PROCESS task is the controlsystem instance of the robots physical movement (MOTIONCONTROL module [2.3.2]).

3. THE DEBUG PROCESS is responsible for the communication between the robot and ROBOTCONTROL (2.4 on page 10) and handles debug messages.

2.3.2 Module concept

In GT2004 different problems are separated into modules. Each module describes a set of tasks, for example there is a module for processing image information and one for controlling the behavior. Because every module has a well-defined interface, different solutions can be implemented for each module and these solutions are switchable at runtime [5]. Figure 2.2 gives an overview over most of GT2004 modules represented by rectangles. Between modules data dependencies are indicated by arrows. These dependencies mean that one module processes the output data from a previous module. Data objects are shown as ellipses.

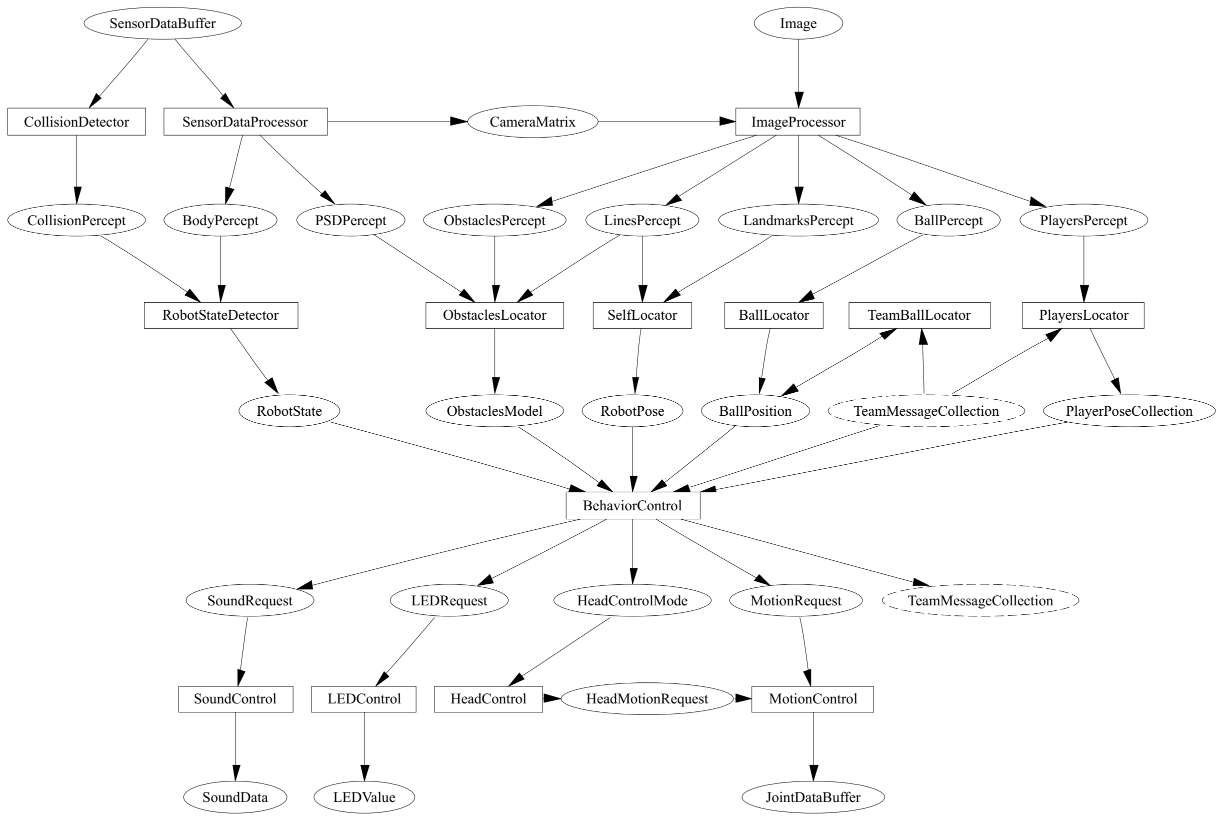


Figure 2.2: Overview of GT2004 modules and data dependencies between them

For example the BallLocator needs data to calculate the ball position. The needed data is combined in a so called BallPercept. This percept is generated and allocated by the ImageProcessor.

Module overview

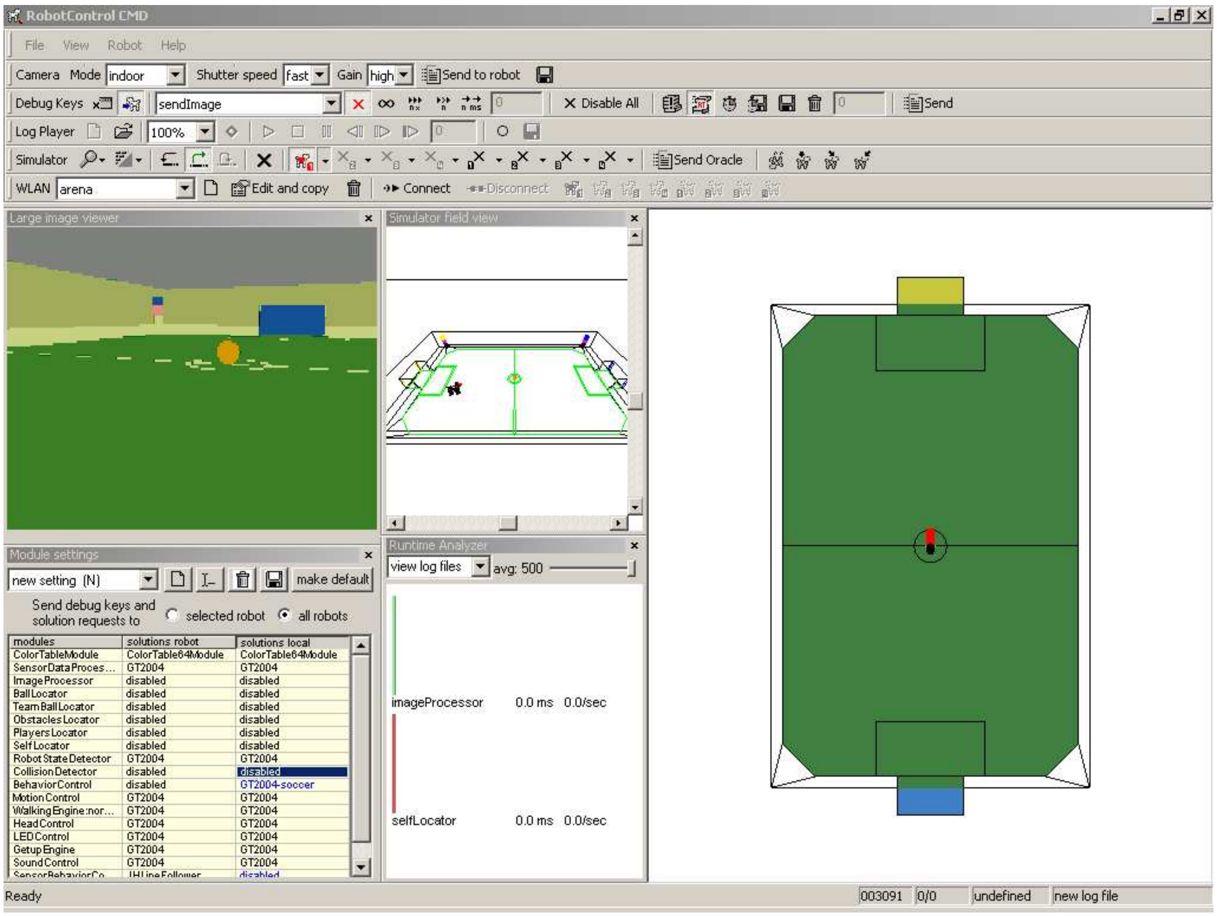
The GT2004 module concept (see chapter 2.3.2)is sufficient for solving the entire task: playing soccer. A general overview of modules and their task follows. [compare Figure 2.2]

- **ImageProcessor:** recognize objects in camera images and calculate their position in a robot-cetric reference system.
- **SensorDataProcessor:** collects sensor (other than camera) information, combine and pre-calculate them into datapackages, called “percepts”. (e.g the CameraMatrix is calculated from several joint-angles and describes the relative position of the camera to the body)

- **RobotStateProcessor:** generates the RobotState which represents the current state the Robot is in. The Data includes for example which button is pressed or what the position of the leg joints is.
- **SpecialVision:** this module performs similar tasks as the imageProcessor, but normally is not in use. Its for special tasks like processing picture information which is not directly linked to playing soccer, like reading a barcode.
- **CollisionDetector:** tests if the robot has a collision with an object or obstacle.
- **BallLocator:** transforms data from BallPercepts to absolute field coordinates, taking sensor noise into account.
- **TeamBallLocator:** Combine a set of percepts received from all teammates BallLocators into a single ball hypothesis.
- **PlayersLocator:** calculates the field coordinates of seen robots from data provided by the PlayerPercept
- **SelfLocator:** calculate the position the robot itself stands on the field, in own field coordinates.
- **ObstaclesLocator:** locates Obstacles on the field and calculates their positions for other modules.
- **BehaviourControl:** takes the current available information about robots and enviroment to decide on how the robot has to act.
- **HeadControl:** controlling and timing of head motions.
- **LEDControl:** turns LEDs (light emitting diodes) on and off.
- **MotionControl:** controls all actors/servos of the robot.
- **WalkingEngine:** calculates sets of joint angles and motor drive speeds to create a walking motion. This is a submodule of MotionControl.
- **SpecialActions:** calculates sets of joint angles and motor drive speeds to create special motion sequences(e.g. kicks, chapter 3.3.2 on page 18).
- **SoundControl:** processes and plays sounds.

2.4 RobotControl

RobotControl is a tool, which is primarily used for the debug communication to the robots. It is possible to establish a connection to one robot, or to all at the same time, with the goal to get the data, like camera pictures, joint angles etc., from the robots.


RobotControl CMD
File View Robot Help
Camera Mode indoor Shutter speed fast Gain high Send to robot
Debug Keys sendImage Disable All Send
Log Player 100%
Simulator Send Oracle
WLAN arena Edit and copy Connect Disconnect
Large image viewer
Simulator field view
Module settings
new setting (N) make default
Runtime Analyzer
view log files avg: 500
Ready 003091 0/0 undefined new log file

Send debug keys and solution requests to ( ) selected robot [✓ all robots]

modules	solutions robot	solutions local
ColorTableModule	ColorTable64Module	ColorTable64Module
SensorDataProces...	GT2004	GT2004
ImageProcessor	disabled	disabled
BallLocator	disabled	disabled
TeamBallLocator	disabled	disabled
ObstaclesLocator	disabled	disabled
PlayersLocator	disabled	disabled
SelfLocator	disabled	disabled
RobotStateDetector	GT2004	GT2004
CollisionDetector	disabled	disabled
BehaviorControl	disabled	GT2004-soccer
MotionControl	GT2004	GT2004
WalkingEngine:nor...	GT2004	GT2004
HeadControl	GT2004	GT2004
LEDControl	GT2004	GT2004
GetupEngine	GT2004	GT2004
SoundControl	GT2004	GT2004
SensorBehaviorCo...	JHLineFollower	disabled

```
imageProcessor    0.0 ms  0.0/sec
selfLocator       0.0 ms  0.0/sec
```

Figure 2.3: Screenshot of RobotControl, the application which is used for debugging

2.5 Main focus of the GermanTeam

The main focus of the GermanTeam is to improve the robots´ ability to play soccer. To do so, the four member universities (Bremen, Berlin, Darmstadt and Dortmund) of the GermanTeam are working at one single project. Until the GermanOpen, which takes place at the beginning of the year, each team is working autonomously, trying to improve their own gameplay and performance on the base of last year‘s GermanTeam code. After the GermanOpen the new improvements and developments of every team are merged, the best candidate for every single module is selected. After this consolidation the four teams are working on the new GermanTeam code with the goal to win the annual RoboCup (world championship).

Chapter 3

Tuning for the ERS-7

In October 2003 Sony introduced a new model of Aibo robots, the ERS-7. It's the successor of the ERS-210 which was used by our preceding project group and also was the model we started working on.

We received our first ERS-7 in January 2004. As we had decided to participate in the GermanOpen 2004 (the GermanOpen competition is annually taking place in Paderborn) with the new robots we had to port the existing software to this new model. The differences between the ERS-210 and ERS-7 can be divided into two major sections: the new hardware of the robot and the new SDK (Software Development Kit) provided by Sony for the new robots.

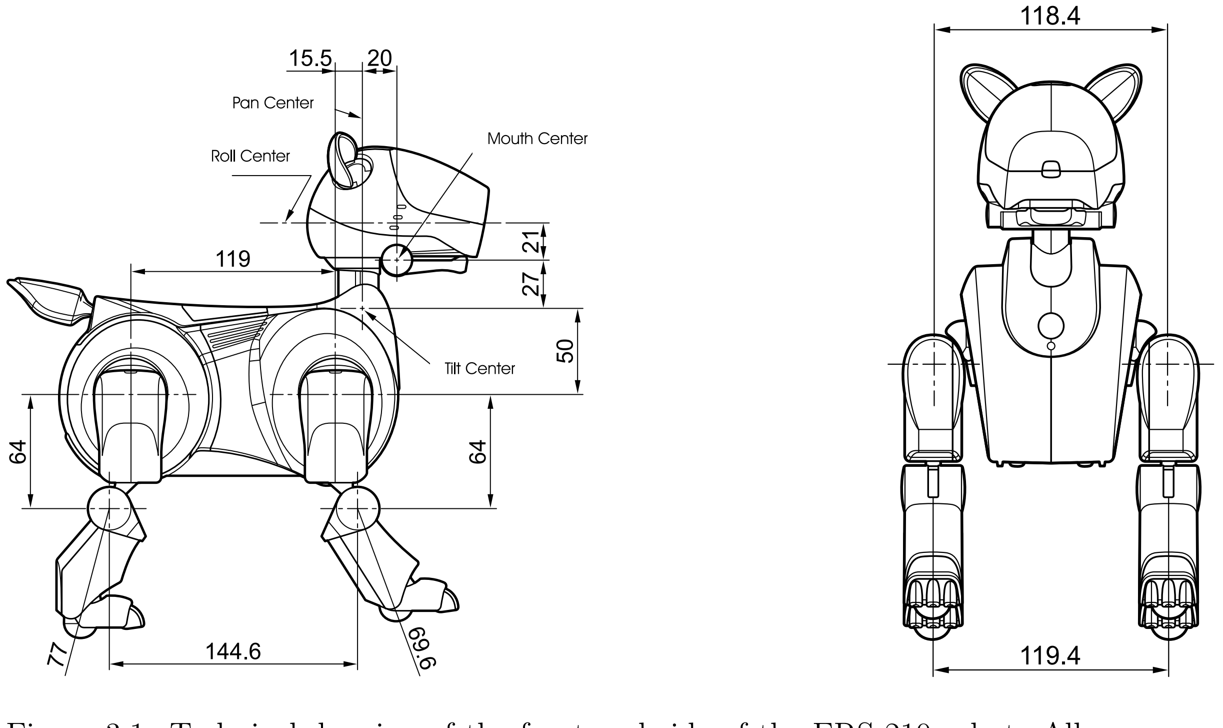


Figure 3.1: Technical drawing of the front and side of the ERS-210 robot. All measurements are given in mm.

The new hardware and software forced us to develop a new walking gait and new kicks which will be described later in this chapter.

3.1 New hardware

The hardware changes Sony made for the ERS-7 consist of two major categories.

1. Physical appearance
2. Internals

As for the physical appearance the main difference of the ERS-7 compared to the older robot is its bigger and heavier body. For example the extremities are about 1 cm longer (as can be seen on fig. 3.1 and fig. 3.2) than those of the old robot.

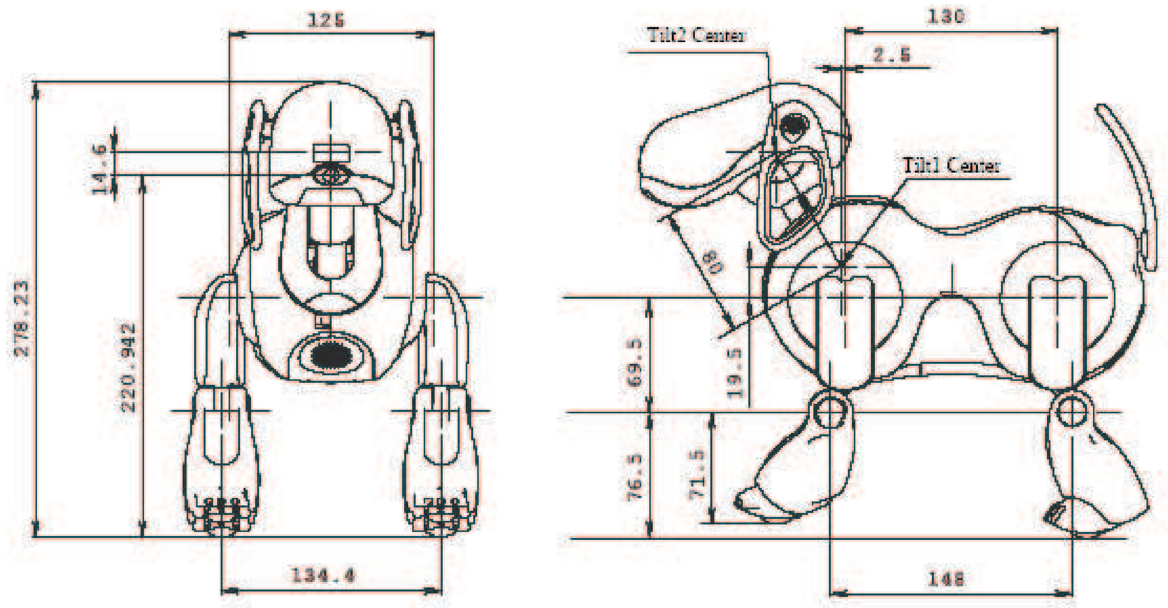


Figure 3.2: Technical drawing of the front and side of the ERS-7 robot. All measurements given in mm.

Also the head is a lot bigger and heavier than in the ERS-210 robot which gives the robot a completely different barycenter. Since Sony gave the ERS-7 a completely different shape, everything concerning the robot interacting with its environment (e.g. walking, handling the ball etc.) had to be redesigned.

Although the number of joints stayed the same, some of them changed in function or position. Not all of the robots joints are relevant for robot soccer. (e.g. the newly designed tail joints are not used for game play in RoboCup) but some changes were more important. The biggest changes that have an impact on RoboCup concerns are the head joints. Like in the ERS-210, there are three joints for the head motion. On the ERS-210 these were: one pan joint (located inside the head), one roll joint (also located inside the head) and one tilt joint (located inside the robot´s body core). On the ERS-7 the roll joint was replaced by a second tilt joint (located inside the head). Also the position of the pan joint was changed (from inside the head (ERS-210) to inside the robot body core (ERS-7)).

Also the motors moving all the joints of the ERS-7 have significantly more power then their equivalent in the ERS-210. Sony did not provide any specification sheets for that but our tests have shown it.

Apart from the joints the button interface was changed, too. The ERS-210 had two buttons on the head and one button on the back. All of them were physical buttons which means that they had to be pressed.

The ERS-7 has four buttons: one on the head and three on the back; all of them are electro-statical buttons which means they only have to be touched.

Sony also did some work on the robot's computer core. The ERS-7 has 64 megabyte of physical memory which is twice the amount of RAM compared to the ERS-210A. The processing power was also improved for the ERS-7, it now has a 576 MHz CPU compared to the 384 MHz CPU of the ERS-210A. The W-LAN(802.11b) which was optional on the ERS-210 is now built-in.

According to the technical specifications the camera was improved on the ERS-7. The resolution of the old camera was 176 x 144 pixels on the UV channels (color information) and 352 x 288 on the Y channel (brightness information). The resolution of the new camera is 208 x 160 on the UV channels and 416 x 320 on the Y channel. Unfortunately, this improvement is only of limited advantage, as the camera also introduced previously unheard of problems like lens distortion and insufficient color correctness, which in the end renders the new camera's images worse than those of the old camera.

3.2 New SDK

Since the introduction of the ERS-7 into the team it showed problems in frequent shutdowns. One of the main causes was "jamming" which means a malfunction in the joints of the robot. This occurs when the robot tries to address a joint angle which cannot be obtained physically. The ERS-7 has a built-in protection to prevent the robot damaging itself. This protection is called JamDetection. The new SDK offers two new methods to control the JamDetection. The first method is the notification of a JamDetectionThreshold. When the default JamDetectionThreshold value is too strict, the programmer can add the following line to the file `VRCOMM.CFG` in the folder `/OPEN-R/SYSTEM/CONF/`:

```
JamDetectionHighThreshold
```

But this did not solve the shutdown problems. Another method which unfortunately includes the risk of damaging the robot is to delete the `EmergencyMonitor` from the code for the robot. The `EmergencyMonitor` controls all processes of the robot and is the protection against the robot damaging itself. It is responsible for any emergency shutdown. Deleting this monitor solves the problem of the shutdowns of jamming, but any other protection against problems like battery overcurrent is also deleted. This method requires careful controlling of the robot by its user to prevent the robot from any damage.

3.3 Software changes

The new hardware forced us to change the software in all modules where modified hardware like joints or sensors is used. On some solutions this is reflected only in some parameter tuning or changes, but on others there was more work to do.

3.3.1 Development of a new walking gait

As far as new hardware was concerned, a new walking gait was required. The *InvKin-WalkingEngine* was used to develop such a gait (see chapter 3.9.1 in the GT2003 Team-report [5]).
Basically a walk consists of several foot positions resulting from a given parameter set of joint angles for each leg. The actual walk is based on a rectangular shape which seperated a movement cycle into 4 phases (see fig. 3.3):

1. ground phase
2. lifting phase
3. air phase
4. lowering phase

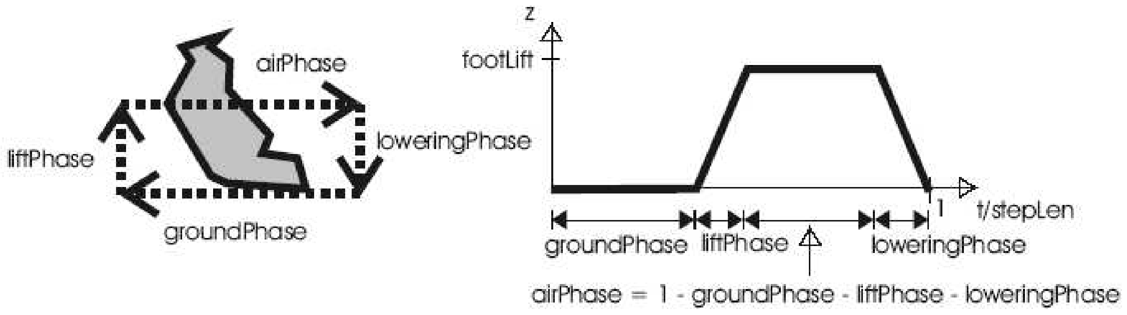


Figure 3.3: Step cycle: on the left side one can see how the different phases belong to the step cycle of the robot. On the right side one can see the timing of the cycle.

The aim is to find a parameter set which allows a robot to move fast from one point to another. For the ERS-210 useful parameters have already been found, but used on an ERS-7 they were completely useless.

In order to find appropriate parameters quickly, the (1 + 1) evolution strategy was used. It belongs to the family of the $(\mu + \lambda)$ evolution strategies. μ is the number of parents from which λ offsprings are generated. So in our case we started with 1 parent, created 1 offspring and then we compared them with a fitness function F. A fitness function is a function which is used to select individuals for mutation and crossover in the next generation.
The evolution strategy we used was also equipped with a self adapting mutation strength, the so called 1/5th-rule, which means that on average 1 of 5 offsprings should be better than its parent. If more offsprings are better, the mutation strength will be risen. Instead if the amount of weaker offsprings increases too much, the mutation strength will be lowered. For further information see [13].

A movement is basically controlled by the 4-Tuple $[dx, dy, d\theta, dt]$. dx describes the movement progress to the front or to the back, dy the progress sidewards, $d\theta$ indicates the degree of movement around the vertical axis of a robot, and dt specifies the time in which each of the three movements should have been done (see fig. 3.4).

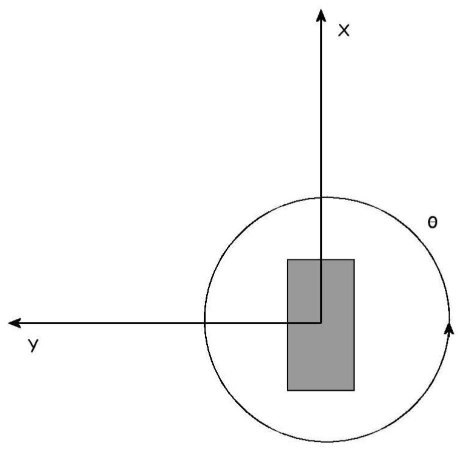


Figure 3.4: General movement of a robot

There have been two approaches to develop a new walking gait. In both the robot starts from one goal point and tries to reach the opposite goal point in a specified amount of time. A goal point is defined as the center point of the goal line.

First Approach

In the first approach, the robot corrects his walk direction using its self locator, so the movement was described as $[dx, dy, d\theta, dt]$, while dy and $d\theta$ depend on the directional correction of the robot (see fig. 3.5(a) on page 18). Due to the fact that the hardware of a robot is not placed absolutely symmetric, the center of gravity is not perfectly in the center of a robot. For this reason a robot will always have to correct its direction during a walk.
In this approach the fitness of a parameter set is mesured by the distance g reached at the end of a walk. So the fitness function of the first approach is:

$$F = g \tag{3.1}$$

The predefined time in both approaches is chosen in a way that the robot reaches the opposite penalty area due to the fact that the self locator has an accuracy of about ±10 cm and the friction of the ground might not be constantly the same on the whole field. So a longer distance leads to minor errors in the measurement.

`procedure of the first approach:`

The robot:

1. aligns at a goal point looking straight ahead to the opposite goal,
2. walks ahead while localizing and correcting its direction,
3. stops after a predefined amount of time and localizes,
4. measures the reached distance and compares it with the distance reached by the parameter set of the parent generation,
5. moves to the nearest goal point and restarts.

Second approach:

The main idea of the second approach is to achieve a fast walk without any correction of the direction, so the movement is given as $[dx, 0, 0, dt]$ (see fig. 3.5(b) on the following page). The fitness function F consideres the distance g and, in difference to the first approach, the deviation h to the real straight walk too. So the fitness function of the second approach is

$$F = g - \beta * h \tag{3.2}$$

where β weights the straightness when calculating the fitness of a parameter set, which allows to define the importance of the deviation during the evolution.

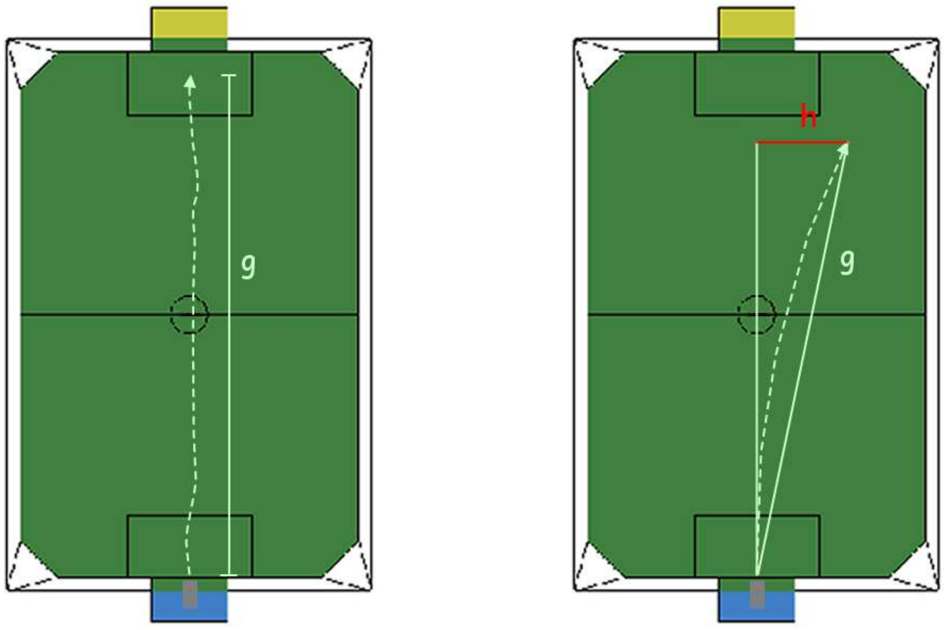
`procedure of the second approach:`

The robot:

1. aligns at a goal point looking straight ahead to the opposite goal,
2. walks ahead without any correction of the walk direction,
3. stops after a predefined amount of time and localizes,
4. measures the reached distance and compares it with the result of the parent,
5. moves to the nearest goal point and restarts.

In both approaches a parameter set of an offspring, which is better than its parent, is evaluated 2 - 3 times to ensure that it is worthwhile to proceed the evolution with that new parameter. If this is really the case, a new generation starts and the last offspring becomes a parent.

Due to the fact that the first approach allows more than only one robot to walk on the same playing field at the same time, which is not the case in the second approach, where a collision cannot always be prevented without intervention by external control instances like for example a human, the *Microsoft Hellhounds* mainly focused their evolution on the first approach and achieved a 34 ±1 cm/s walk.



(a) First approach (b) Second approach

Figure 3.5: Different evolution approaches

3.3.2 New kicks and MOFs

Because of the changes to the ERS-7, i.e. its modified physical dimensions (see chapter 3.1 on page 13), the 2003 versions of all movements were rendered ineffective: the kicks and catches which require the ball to be in front of the Aibo failed, while the kicks which need the ball to the side of the Aibo worked, but only barely. These had to be tuned and new moves for the kicks with the ball in front of the Aibo had to be designed.

That is why we created new kicks. Normal kicks are motion files from now on referred to as "MOF" files, due to their .mof file extension. There are other possibilities to kick the ball. For example a head control (see chapter 2 on page 6) mode can be created, which lifts the head up, turns it to a side, then takes the head down and turns it to the other side. If a ball lies in front of the Aibo, it can kick the ball with such a head control mode. Or the Aibo can simply run against the ball, inducing its momentum onto the ball.

For the GermanOpen 2004 (see chapter 7.1 on page 77), only MOF kicks were used, but motion files can be used in other situations as well. For instance, while we created a number of new kicks, we couldn't find suitable movements to be able to kick the ball into every direction desired. So, we decided to let the robot approach the ball, then let them turn until they reached an angle to the ball which would allow one of our kicks to move the ball into the desired direction. Initially, we tried to do the turning via our motion engine. Unfortunately though, we quickly realized that this engine was not precise enough without visual input, which could not be provided, since the Aibo´s head in these circumstances was already over the ball and could not see it (see fig. 3.3.2 on the following page). Additionally, this approach would cause serious maintenance work every time we incorporated a new set of walking parameters. So, we approached this problem with mofs as well. We designed mof parameters which would turn the robot around a fixed point by 30, 60, 90, 120 and 180 degrees, respectively. This seemed to work well in our test games

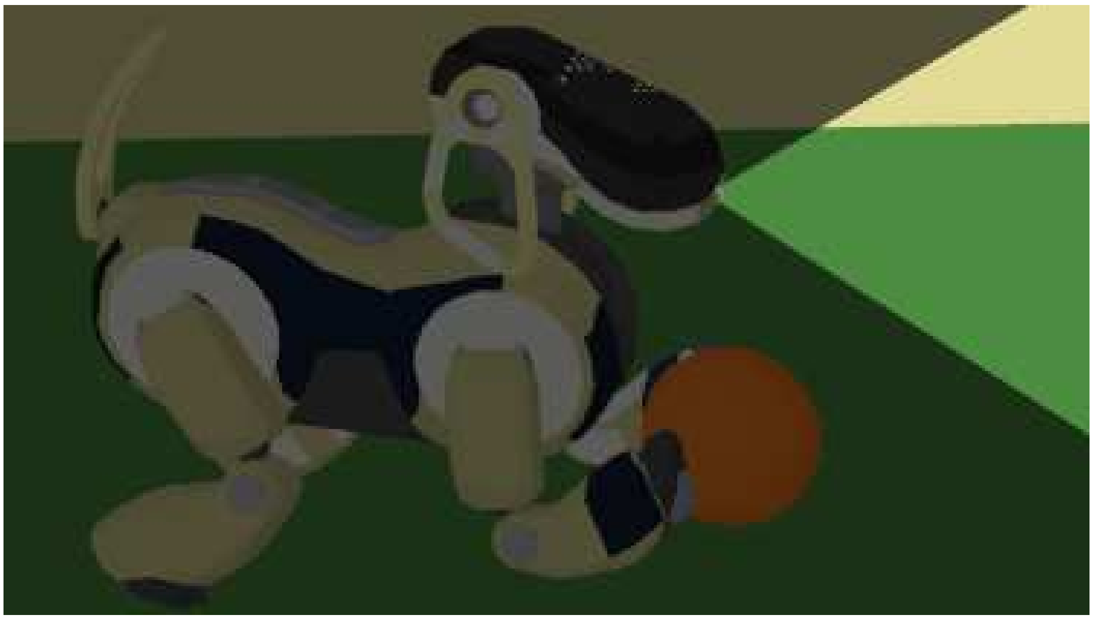


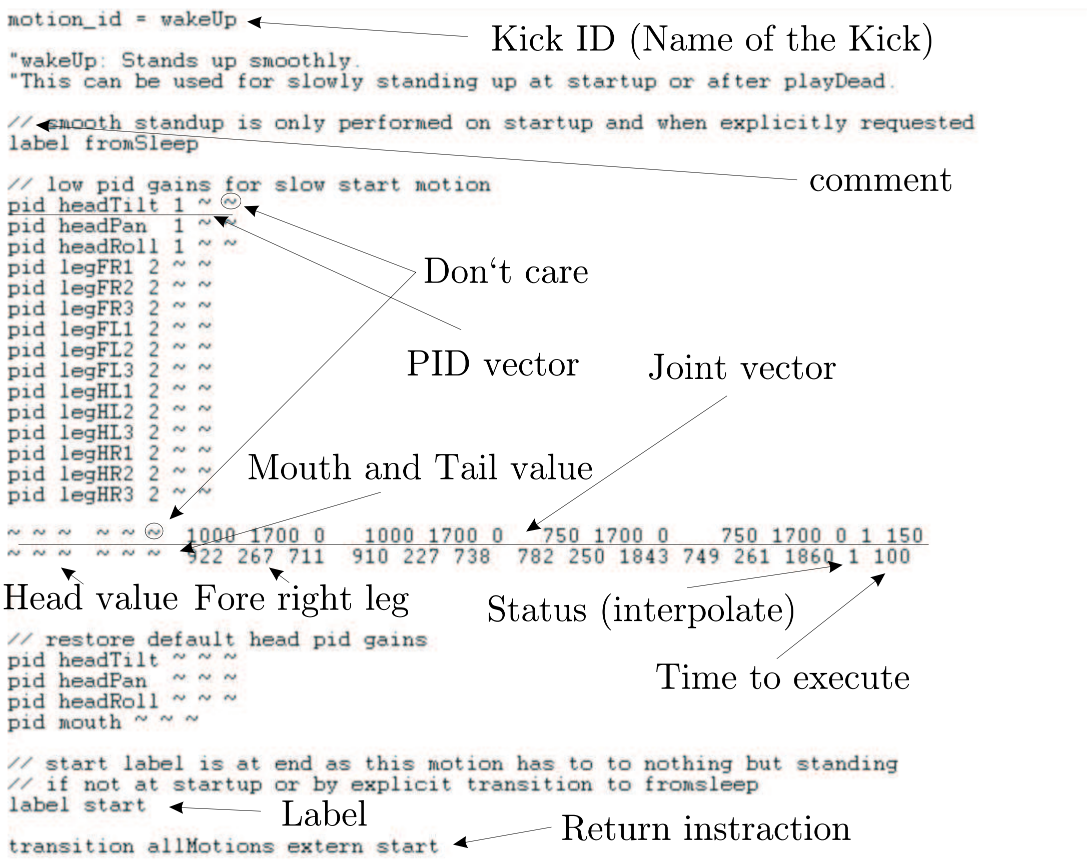
Figure 3.6: An Aibo which turns around a ball and cannot see it. To illustrate this the opening angle of the camera is lit.

against the ERS-210 robots. Either, the new robots were fast enough to have the turn completed, before any interfering ERS-210 would arrive and even if the ERS-7 didn't have the time, it would have enough power to push the smaller ERS-210 out of the way and complete its turn. However, in the first games against other ERS-7 robots, this didn't work as the ERS-7s wouldn't be pushed away and the kick failed. So, we eventually decided against the strategy of turning around the ball via mof special actions.

We also used motion files for cheering (see chapter 8 on page 85).

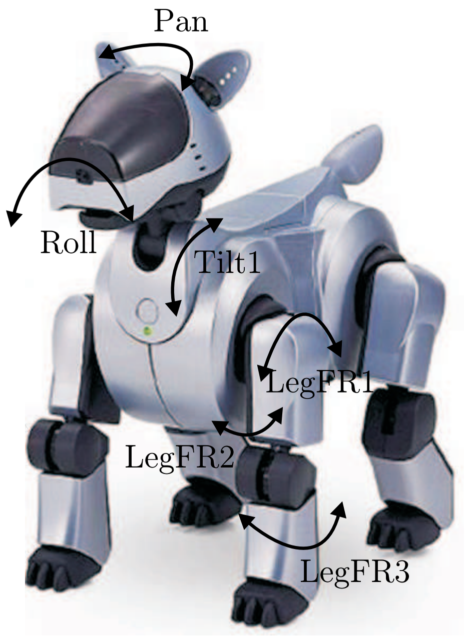
MOF file description

Short patterns of motion are written in files. Such files are called mof (motion file) and have the extension "*.mof*". One such example can be seen in figure 3.3.2 on the following page. In the first line of the file, a name must be specified via the motion_id keyword. Every normal kick has one label called start, these labels are used as entry points. More than one label can be defined to create several entry points. The default entry point of a mof is specified in the file "extern.mof". The last line of a mof contains the return instruction: *"transition allMotions extern start"*. Between the entry point and the return instruction there are so-called "motion vectors". There are two kinds of vectors, *joint* and *pid* vectors. Via the pid vectors pid values of a servo gain are set (see capter 2 on page 6 or fig. 3.8 on the following page). The line starts with "pid", to mark the line as a pid vector. Then the p, i and d values need to be set. This kind of vector is only used in wakeup motions, to switch off and reset the joints (see fig. 3.3.2 on the next page). If a value is irrelevant, it is marked as "don't care" with "~". More important is the joint vector (see fig. 3.9(a) on page 21), which is used in nearly every kick. It is a sequence of 20 values, 18 joint, 1 status, 1 time (for don't care the "~" is used, too). Time means the delay time until the next vector can start. The status value determines whether the joint movement is interpolated over the time value (value = 1) or as fast as the joint servos allow (value = 0). The first three values of the joints represent the head joint values, the first is the headTilt1, the second the headPan and the third the headTilt2. The next

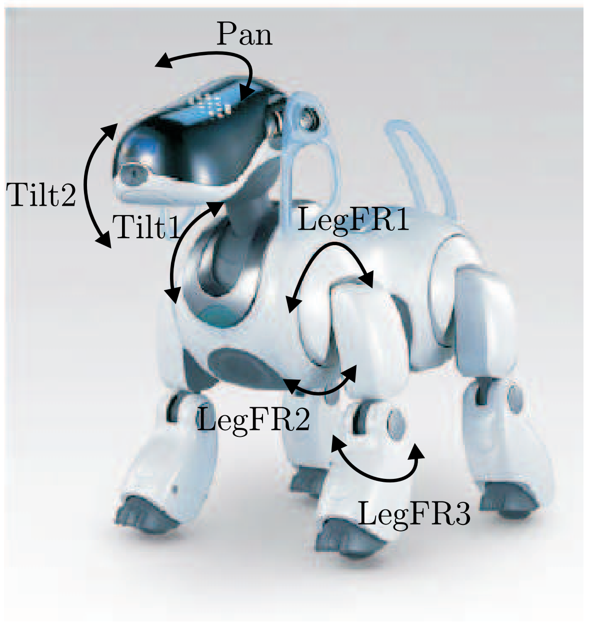


```
motion_id = wakeUp

"wakeUp: Stands up smoothly.
"This can be used for slowly standing up at startup or after playDead.

// smooth standup is only performed on startup and when explicitly requested
label fromSleep

// low pid gains for slow start motion
pid headTilt 1 ~ ~
pid headPan  1 ~ ~
pid headRoll 1 ~ ~
pid legFR1 2 ~ ~
pid legFR2 2 ~ ~
pid legFR3 2 ~ ~
pid legFL1 2 ~ ~
pid legFL2 2 ~ ~
pid legFL3 2 ~ ~
pid legHL1 2 ~ ~
pid legHL2 2 ~ ~
pid legHL3 2 ~ ~
pid legHR1 2 ~ ~
pid legHR2 2 ~ ~
pid legHR3 2 ~ ~

~ ~ ~   ~ ~ ~  1000 1700 0   1000 1700 0   750 1700 0     750 1700 0 1 150
~ ~ ~   ~ ~ ~  922 267 711  910 227 738  782 250 1843 749 261 1860 1 100

// restore default head pid gains
pid headTilt ~ ~ ~
pid headPan  ~ ~ ~
pid headRoll ~ ~ ~
pid mouth ~ ~ ~

// start label is at end as this motion has to to nothing but standing
// if not at startup or by explicit transition to fromsleep
label start

transition allMotions extern start
```

Figure 3.7: “wakeUp” mof as example for a motion file

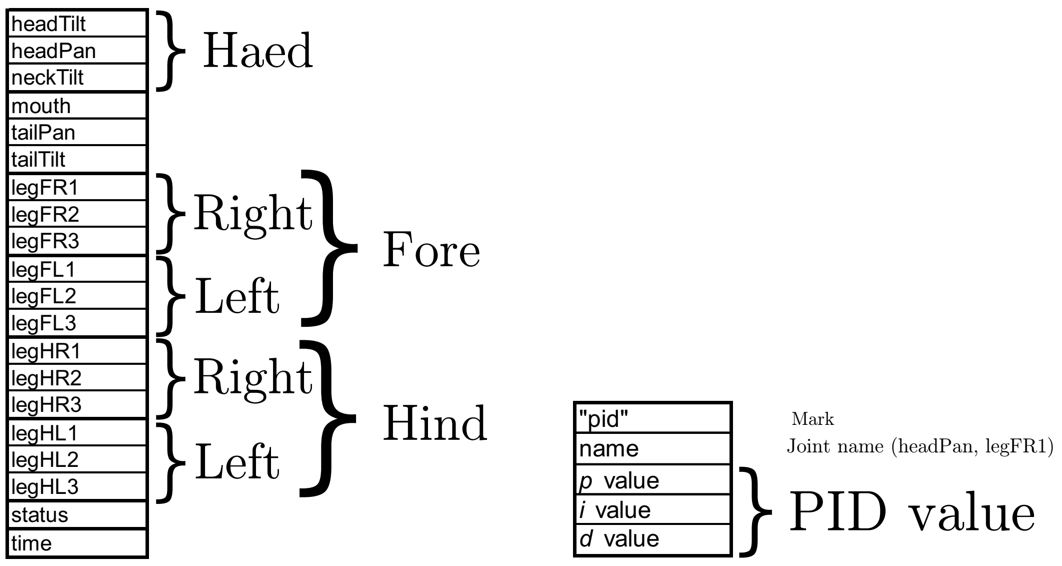


(a) Overview of the joints of a ERS-210



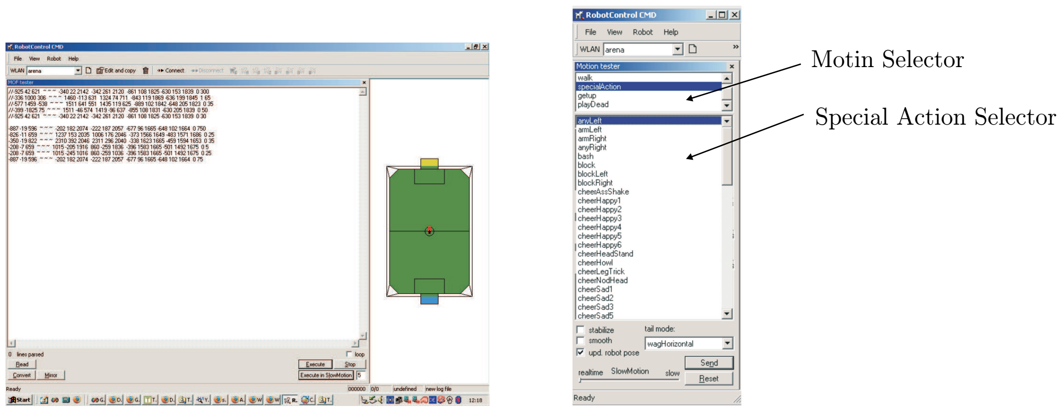
(b) Overview of the joints of a ERS-7

Figure 3.8: overview of the joints of ERS-210 and ERS-7. The roll joint of the ERS-210 has changed to a tilt2 joint of the ERS-7



(a) the format of the joint data vector (b) the format of the pid data vector

Figure 3.9: overview of the type of data vectors of a mof file.



(a) Overview of Mof Tester Dialog with BashPrecize as example for a motion (b) Overview of the MotionTester dialog

Figure 3.10: mof and motion tester dialog

value is for the mouth and the next two are for the tail. Then four triplets follow, one triplet for each leg, the first leg is the front left one, the next is front right, then behind left and behind right. The values of the triplets are for the joints from core to paw (see fig. 3.3.2 on the previous page). To make a motion file more readable, comments may be inserted, which will start with two “\”.

Tools

RoboControl has two dialogs, used to create new motion files, called “MOF tester” and “Motion tester” [5]. The “MOF tester” dialog (see fig. 3.10(a)) has an edit box and 6 buttons:

- Read
- Execute

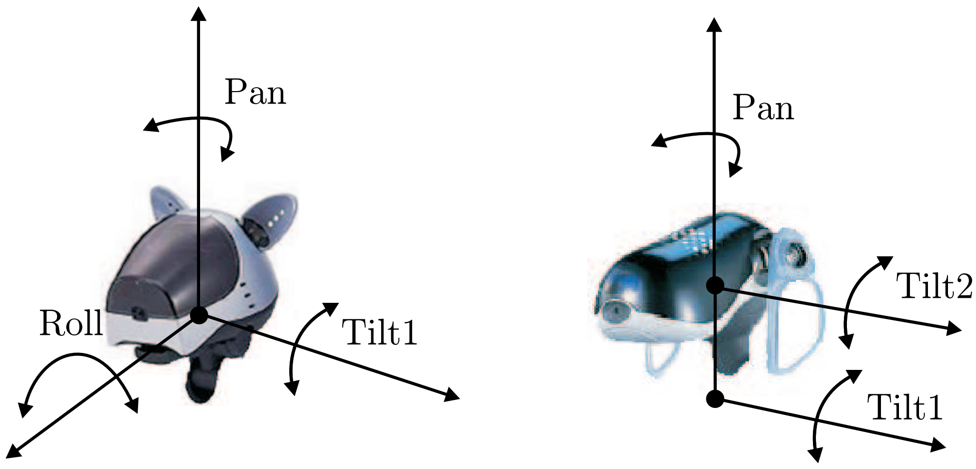


Figure 3.11: 3 headjoint values pan tilt and roll. The roll joint of the ERS-210 had changed to tilt2 in the ERS-7

- Execute in SlowMotion
- Stop
- Convert
- Mirror

The “Read” button reads the current joint values of the connected and inserts the joint vector into the edit box, the “Execute” button sends a motion request with the selected joint vectors from the edit box to the robot, and the button called “Execute in SlowMotion” multiplies a delay value from the editbox right next to the button to the motion request before it send it. The “stop” button immediately stops the Aibo’s motion, the “convert” button converts the motion into raw data which is a format that can be placed in the source code and the “mirror” button switches the right side joint values to the left and vice versa, this can be used to mirror a kick to the left easily to get a exactly same kick to the right. At the beginning of the project group, there was a conflict with the head joint values. Due to the ERS-7 head only having the pan joint moving sidewards (as opposed to the ERS-210 head having two, the pan and the roll) (see fig. 3.3.2) updates to the “MOF tester” had to be made, adapting to this new joint layout.

Using the “MOF tester” requires the use of a bug workaround. Since in the *Debug* solution of the *motion control* module a bug occurs, that won’t allow to reset the joint gains, the Aibo needs to be booted with the *Default* solution and then switched to *Debug*. This ensures the Aibo performs its “getup move”, which automatically resets the gains, before switching to debug mode.

This bug has not yet been fixed since it is deeply rooted in the framework. Since for testing MOFs there needs to be a connection established with robotcontrol anyway, this does not cause much overhead complications.

With the “Motion tester” dialog (See fig. 3.10(b) on the preceding page), motion files can be executed. Here they are called special actions. The ”Motion tester” has a combo box, a *send* and a *reset button*. In the combobox, the motion type will be selected for special actions such as *specialAction* and a new combobox will appear, with all available special actions. The selected *specialAction* will be executed in a loop from a send command until the *reset* button is triggered.

To execute a motion with the motion tester, it must be registered in the code as a special acion, the *default* solution has to be selected in the *motion control* module and the behavior must be *disabled* (as it would overwrite any motion request otherwise). To register a motion, the joint vectors must be saved in a file in the mof directory, the MotionRequest.h and the extern.mof must be updated.

Tuning a mof

There are several ways to tune a movement, it can be made faster, stronger or more precise. It is difficult to tune a movement though, one problem is that often a faster kick is softer or less accurate. On the other hand an opponent can disturb a slow shot more easily. It might place itself in the shot path or push against the robot, thus interfering with and possibly destroying the entire motion sequence.

The “MOF tester” cannot be used to tune a kick, because if the DebugMotion is running, the engines are weaker and slower, therefore, for each change it is necesary to compile and create a new memory stick. This would take a lot of time, however, this can be improved with a small trick: To test modified versions of one motion, files from other motions can be used. So more than one change can be tested with one compilation. The “overwritten” motions should be carefully backed up though, so they can be restored, once the modified mof is finished.

Before movement tuning can start, the movement in question must be thoroughly analysed, tested in several scenarios and setups and all observations should be noted meticulously. Often a change makes the motions better at some scenario while it weakens the motion in another. We selected scenarios which are important in games and defined what makes a result acceptable. Then we subsequently left out different vector lines of the motion to find out which steps were important for the result and which could be left out. Usually we changed only one joint vector at a time, because it is easier to find the right value that way. Often when we tried to change several joint values, we encountered problems to identify the “correct wrong” value.

For example we have created a new backward kick called “*MSH7NewBicycle*” which is composed of three actions: 1) the catches the ball, 2) it lifts it up onto its neck, and 3) it sits up and the ball rolls down the back. One problem was that the kick would take more than 3 seconds to execute, which would break a rule called *ball-holding* (Rules see section Rules 2.1 on page 6). So the kick was in a first phase tuned that it took less than 3 seconds, but by doing so it became inexact: the ball would roll backward but strew. We considered this as acceptable though, because we created the kick to get the ball away from the border and the kick accomplished that. A second problem though was that if an opponent knocked against our robot, it would lose the ball. This can become a problem, because if this happens near the own goal, the ball might incidentally roll into it; this in fact occured during the AmericanOpen (see 7.3.2 on page 83). We have tried many changes but none has solved this problem. Eventually, we found out, that the Aibo only lost the ball when the opponent knocked from the side where the Aibo wanted to lift up the ball from. That is why we decided to create two kicks, one that lifts the ball over the left side and one over the right side. Despite all these efforts the kick didn’t stand the test of time as the decision of which of the two versions to chose depended on a working opponent robot detection. Unfortunately, this opponent detection could not be provided to date, so we had to dismiss this kick as we did not want to risk scoring own goals. (see 4.3.8 on page 44)

Our self-made MOFs

All the MOFs we created fall into four categories:

1. Kicks that were used during the games
2. Turn movements
3. Cheering/Audience amusement
4. MOFs that were discarded

The first category includes all those MOFs that were in fact used during the games to kick the ball in some direction. The second category includes all turning moves that are used to cover situations, where the robot is positioned inappropriately towards the ball for any of our kicks. These movements align the robot to enable it to use one of the normal kicks. The third category includes all MOFs, that are not suited for ingame use, but are in one way or the other visually impressive, stunning or just too entertaining to be kept from the public. The fourth category includes all those MOFs that ultimately were not used at all, because they didn't work at all, were inferior to other moves of equal kind or rejected because of no use for the behavior because the situations the kick would be useful do not occur frequently enough or require a degree of self-localization which simply cannot be provided by current means.

MOFs we used in the games

The kicks MSH7NewBicycleFromLeft (fig. 3.12), MSH7unswBash (fig. 3.13), MSH7SlapLeft (fig. 3.14 on the following page) and MSH7LeftHook (fig. 3.15 on the next page) actually were used for the games.

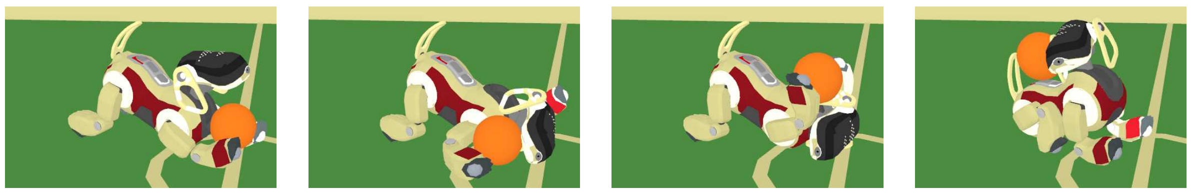

(a) catch ball (b) put ball to right (c) put ball in neck (d) sit and stay up

Figure 3.12: MSH7NewBicycleFromRight

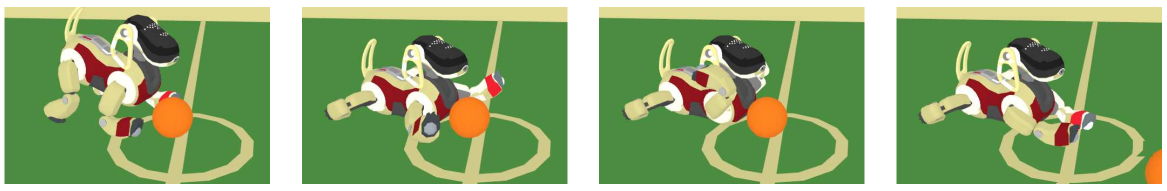
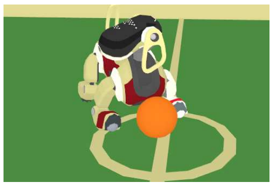
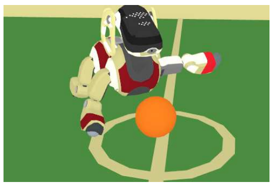
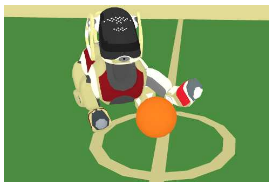

(a) ready (b) catch ball (c) lift arms (d) hit ball

Figure 3.13: The *MSH7unswBash* was taken from last year´s code and adapted to the new robot.


(a) ready


(b) lift arm


(c) hit ball

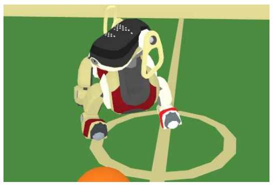
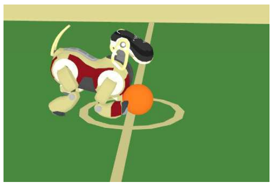
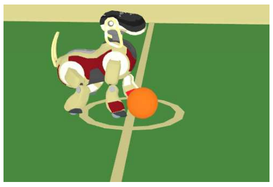
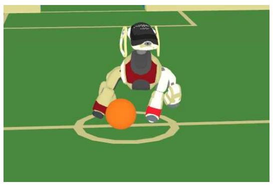

(d) finished

Figure 3.14: The *MSH7SlapLeft* was created to quickly move the ball from the border by hitting it from above with the left arm.


(a) ready


(b) get behind ball (side view)


(c) get behind ball (front view)

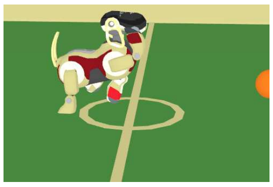

(d) hit ball

Figure 3.15: The *MSH7LeftHook* is a strong forward kick.

Turning MOFs

We created MOFs to turn around the ball for 30,60,90,120,180 degrees to the left or right and called them "MSH7Turn + direction + angle" (for instance *MSH7Left90*), because for a few angles, especially for angles over 90 degrees we did not find any suitable kicks. For the turning motions the ball must lie in front of the Aibo, then the Aibo lifts up and holds it with its front legs, so that the opponent cannot reach the ball, and then turns for the specified angle. We used this for testing only, due to reasons specified above see 3.3.2 on page 18, and replaced them later by a special walking engine *InvKin:MSH2004TurnWithBall* (see 3.3.1 on page 15).

MOFs for cheering/audience amusement

We created some kicks, which are nice to watch, but really aren't of any use for a game, because they are too slow or weak or do not even work the way we intended them to. So we did not use them except for cheering/show-off reasons. Such kicks include the like of *MSH7FakeKickRight/Left*, *MSH7ComplicatedKick* and *MSH7StrangeBackSlow*. Except for these kicks, which were downgraded, we also created motion files for cheering, e.g. for the DemoStick, Chapter 8 on page 85, that were never meant to be used in game, but explicitly created for audience display.

- MSH7FakeKickRight
  The *MSH7FakeKickRight* catches the ball, then hits it with the left paw to roll it to the right paw. Then the right paw hits the ball and only then the ball rolls forward. Slow and unusable for a game, but visually impressive.

- MSH7ComplicatedKick
  If the Aibo executes the *MSH7ComplicatedKick* it will only move one single joint each motion step. This takes a lot of time, but it is nice to watch, because the motion looks like"robot stop motion", again this was used for audience amusement only since it was no good for a game.

- MSH7VanGogh

We also created a motion, which tears the robot's ears off and called it *MSH7VanGogh*. This motion was initially not meant for cheering. As the ears seriously hindered some of our kicks (especially the *MSH7NewBicycle* kicks), we wanted to get rid of the ears. Unfortunately, at game start the robots needed their ears put in place by official ruling (see 2.1 on page 6). So, our only chance of using those kicks was to let the robot remove their ears themselves. Since this action took quite some time (even though it was perfectly legal), officials later allowed all teams to start a game with their ears off, so our vanGogh move later was degraded to cheering/amusement status.

Rejected MOFs

We rejected some of our MOFs for different reasons like the kick not seeming desirable in any scenario. Such a kick is the *MSH710cm*, which reliably moves the ball forward for 10 centimeters, which is of no use since in that case it should rather be dribbled forward. In other cases the kick did not work at all like the *MSH7ABombBehind* (the Aibo often moved the ball in many different unpredictable directions). Some kicks were superceded by other, more efficient MOFs like the *MSH7ForwardLeft* for example.

Chapter 4

Image Processing

The ImageProcessor module is analyzing the image sensor data of the robot. Mainly objects allowed on the playing field are recognized by the image processor [see Rulebook]. So the image processor is the only module that provides input data about the vision of a robot while playing soccer. The legacy image processor from the GermanTeam-Release 2003 is the GT2003ImageProcessor [5].

4.1 Motivation

One of the first tasks of the projectgroup to acquaint itself with the GT-Code was to make several specialists of the GT2003ImageProcessor more scalable. For example the Ball Specialist of the GT2003ImageProcessor used a fixed number of points at the edge of a given orange ball in a frame, which were taken into account to calculate the circle which fits best the seen ball. It was regarded as a good enhancement to dynamically set this value higher or lower depending on free processing time. Nevertheless it was just a hack to enhance the scalability of an existing solution. In order to that it was no surprise that soon the idea was born to implement a new, clean Image Processor from scratch, which would overcome the limitations of the existing ones. A modular concept was demanded, which divided the tasks of the Image Processor in Ball, Goal, Landmark, Field and Opponent detection. In this context the main idea was to give dynamically priority to those tasks, which are most important in a given situation. In order to make this possible the Image Processor would have to have control over each specialist. To guarantee even more scalability the rastersize (i. e. the amount of lines/ rows considered in calculations) and therefore processing time should be adjustable too, not only global for all specialists but individually changeable for each of them. All in all the new Image Processor should be able to dynamically switch between optimal results in a reasonable processing time at the one end, and fast approximative solutions at the cost of accuracy of the detected objects at the other end. With this concept in mind work on the Raster Image Processor (RIP) started.

At the beginning of the project we analyzed the preconditions of our plans to implement a virtual robot playing soccer. We were dissapointed about the self-, ball- and opponent localization. We found out that the most of the accuracy isn't lost in the locator solutions of the German Team Release, but in the image processing solution. Therefor it was our aim to increase the accuracy of the detection algorithms while decreasing their misconceiving. Since one of our main intentions was to implement resource sharing for the robots, we wanted to have the ability to schedule the work of the image processor. The GT2003ImageProcessor did not support this feature and the most of its detection algorithms weren't scalable.

4.1.1 Color Correction

The image provided by the ERS-7 robot isn't provided equally over all the area of the image. Especially in the corners of the image it is too blue and a little bit dark. So we needed a pixel based color correction. This wasn't done by the PG, but by a faculty staff member. He implemented a look up table with correction values for each pixel of the image. With this table every pixel on the image can be corrected. The correction can be individually adapted to every single ERS-7 robot. This is done by some color coefficients as input data for the look up table creation, which are calculated from some test images taken out of several robots. Our intention was to use this color correction directly in the scan process, without correcting the whole image. Since we wanted to use the ColorTableTSL, we had to integrate the color correction in ColorTableTSL-Calibration-Tool. Otherwise the color classification is defective if the robot uses the color correction.

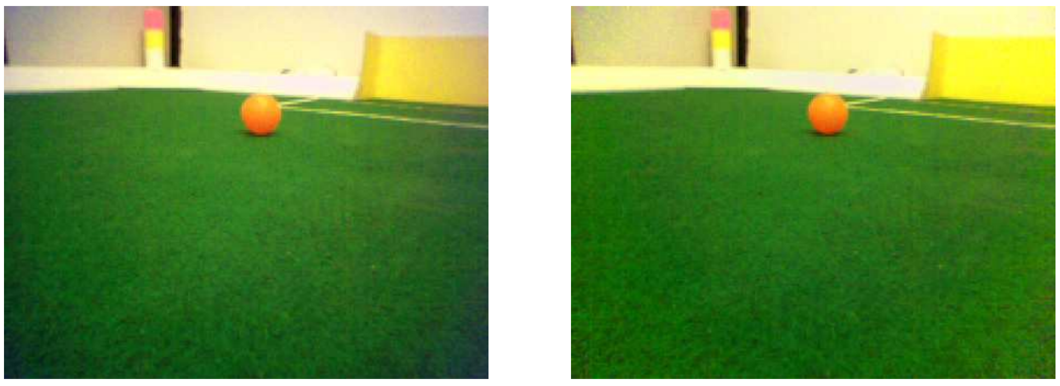

(a) Original image (b) Corrected image

Figure 4.1: Image as seen by an ERS-7 on the playing field

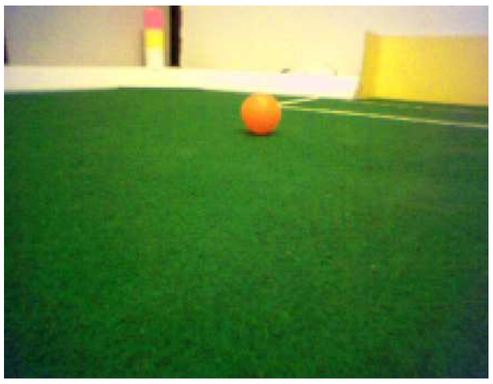
4.1.2 Supporting Color Tables

The ColorTable module represents the classification of several color classes of interest. That means a disjoint definition of semantic colors like black, orange or pink in the colorspace. An example for such a classification is shown in 4.2(b) on the following page.

The *GT2003ImageProcessor* was implemented for use with *ColorTable64* and doesn't support any other ColorTable-module. Since the GermanTeam has more than one implementation of the ColorTable module, we wanted to have a image processor that supports the ColorTable module in general. The two additional solutions for the ColorTable module we wanted to use are the ColorTableTSL[8] and ColorTable32K.

4.2 EdgeDetection

To efficiently detect the shape of objects in an image, we firstly need some feature points of these shapes. We call them *edge points*. Almost every detection algorithm ,we considered


(a) Original image

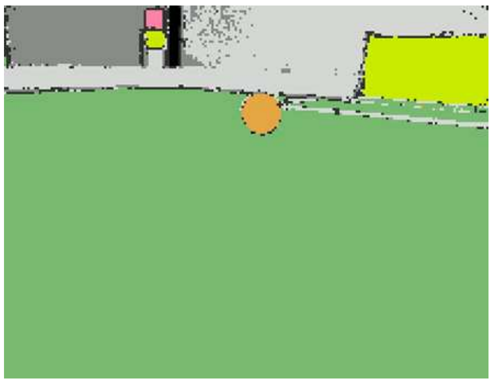

(b) Image classified by ColorTable32K

Figure 4.2: Image as seen by an ERS-7 robot on the playing field.

to implement, needs some pixels of the object's outline. So we wanted to be able to detect these feature points.

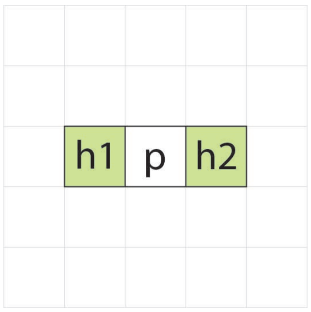
We define *edge points* as pixels that have a large difference of brightness and color compared to their neighbours. To detect such points we wanted to use a simple kind of spatial filter, that generates an amount of *edginess* for a pixel. After a few tests with different filters we decided to use some simple first derivative gradient filters [9] shown in figure 4.2. The image is provided in the YUV colorspace. We use all 3 dimensions to calculate the edginess of one pixel. Let's see how the horizontal edginess e_h, vertical edginess e_v and the cross edginess e_c are calculated:

$$e_h = |h1_y - h2_y| + |h1_u - h2_u| + |h1_v - h2_v| \tag{4.1}$$

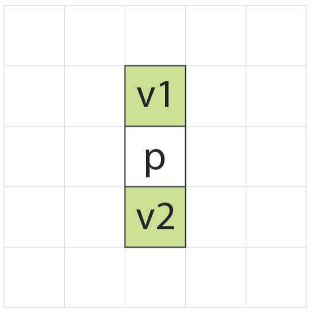
$$e_v = |v1_y - v2_y| + |v1_u - v2_u| + |v1_v - v2_v| \tag{4.2}$$

$$e_c = \begin{cases} e_h, \; if \; e_h > e_v \\ e_v, \; otherwise \end{cases} \tag{4.3}$$

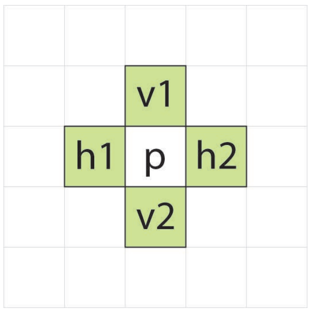
Note, that e_c is usually calculated with $e_c^2 = horizontalgradient^2 + verticalgradient^2$ and the direction of the edge can be calculated with $\arctan(vertical\ gradient/horizontal\ gradient)$. With $\arctan(e_h/e_v)$ we can only differ between horizontal, vertical and square lines.



(a) horizontal gradient filter



(b) vertical gradient filter



(c) cross gradient filter

Figure 4.3: Basic gradient filters - Note, that p is the considered pixel. The green pixels represent the neighbourhood used to calculate the *edginess*.

We combined these filters with a Bresenham line scan [2], the idea of non-maxima-suppression and threshold hysteresis as used in Canny Edge Detectors [6]. This led to a kind of edge scanner that iterates from pixel to pixel in a given direction, while searching for local maxima of their edge votes, if a edge vote is greater than the threshold e_t. This edge scanner can improve the accuracy of detection algorithms and is reusable for every ImageProcessor solution of the GermanTeam. We used the Cohen-Sutherland algorithm for line clipping to be able to determine a starting point of a scan, if a scan line starts beyond the image borders. What would be useful, but is not implemented yet, is to use line clipping for searching the end point of a scan line.

In order to have a filter that detects edges that are aligned to the scan line, we implemented one more filter for the scans, which is shown in figure 4.4. The vote of this filter e_l is calculated analogously to the horizontal and vertical gradient filter.

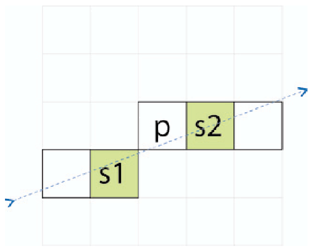


Figure 4.4: Scan line filter - Note, that p is the considered pixel. The green pixels represent the neighbourhood used to calculate the *edginess*. The other highlighted pixels belong to the scan line.

$$e_l = |s1_y - s2_y| + |s1_u - s2_u| + |s1_v - s2_v| \tag{4.4}$$

Some results of a global edge analysis with the *GT2004EdgeDetection*, that is the implemetation of the ideas described above, you can find in figure 4.2 on the next page.

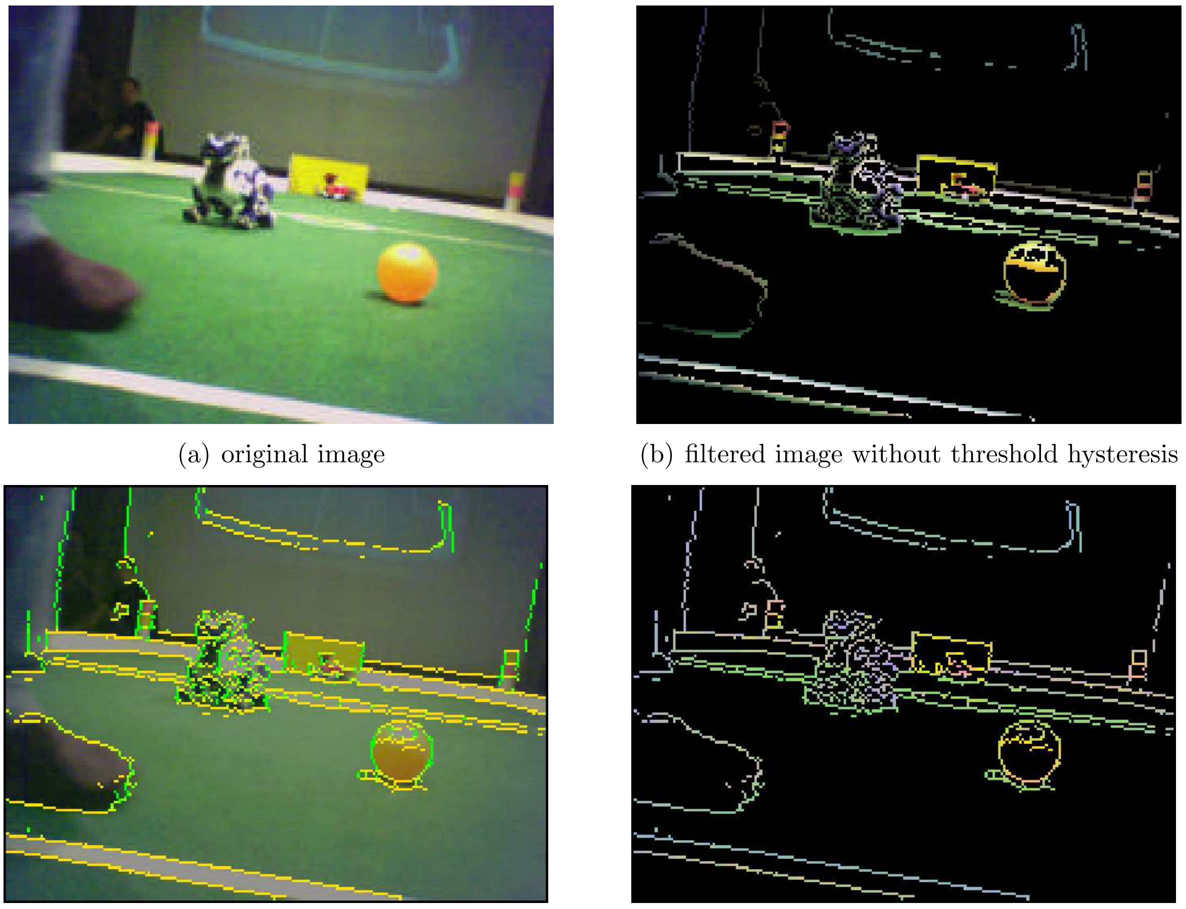
The advantages of this edge detection solution are:

- The performance on blurred images. Only a large amount of blur impacts the edge detection.
- The running time to detect edges is low against global edge analysis.
- Reusable for every solution of the ImageProcessor module.
- Good quality/run time ratio.

The disadvantages are:

- The threshold e_t must be adjusted to the amount of contrast in the image.
- Responses a bit on noise, if the images are too dark.
- Difficult to use on small objects, that are not flat, like far away balls or corner beacons.

Some more examples for the use of this edge detection implementation can be found in section 4.3 on the following page.


(a) original image

(b) filtered image without threshold hysteresis

(c) filtered image with threshold hysteresis - Note, that the detected edge points hold some directional information about the edge they belong to.

(d) filtered image with threshold hysteresis - Note, that the detected edge points hold some color information from the original image.

Figure 4.5: Some examples of global edge analysis with the *GT2004EdgeDetection*. The whole analysis runs with 80Hz on an 1.5GHz centrino laptop, without any optimizations.

4.3 Raster Image Processor

In consideration of our defined intentions we decided to implement a new solution for the module ImageProcessor. It's named RasterImageProcessor, from now on, referred to as RIP. In the following subsections we will give an overview how we planned and implemented the RIP.

4.3.1 Architecture

The most important intentions for the architecture were compatibility to the German Team code and the reusability of the new implementation. Since image processing is a difficult task, we thought about a redundant and modular class concept. For example it should be possible to have different detection algorithms for the same kind of objects without rewriting the whole image processor solution.

Now we give a brief description of the classes the RIP actually consists of:

- The class *RasterImageProcessor* is an extension of class *ImageProcessor* and a context for the specialists and the strategy. It provides the image processor interfaces and extensions to them like color correction or the calculated horizon line. It also holds a collection of specialists and one strategy. The strategy is executed every frame.

- The class *RasterStrategy* is an abstract base class for all strategies of the RIP. An implementation of this class decides what pixels are scanned and delegates a collection of specialists. It also has to provide the input data for the several specialists (e.g. runs, feature points or even clusters of pixels or runs) it delegates.
- The class *RasterSpecialist* is an abstract base class for all specialists of the RIP. An implementation of this class should provide a detection algorithm (e.g. player detection, beacon detection).

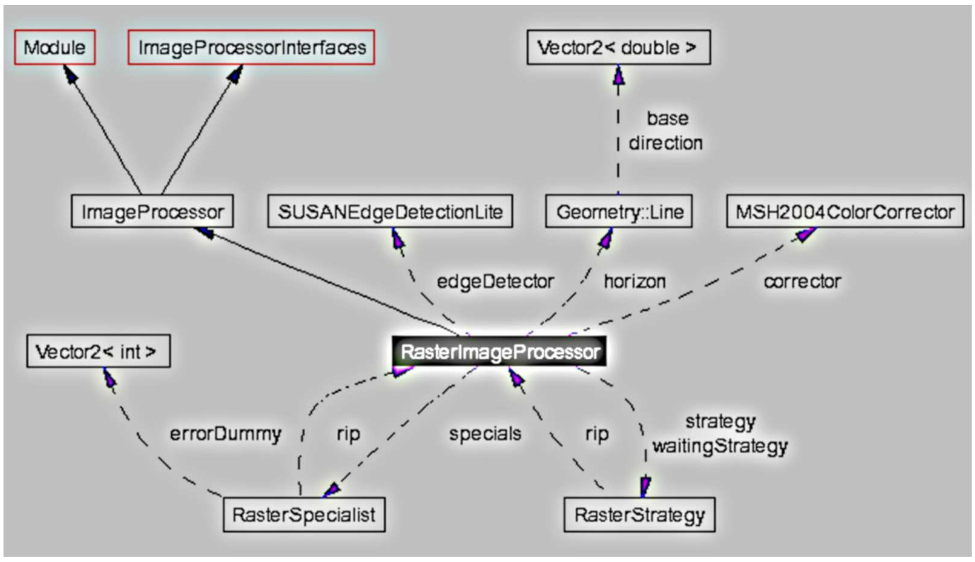


Figure 4.6: Collaboration diagramm for class *RasterImageProcessor*.

- The class *RDefaultStrategy* is the default implementation of a *RasterStrategy*. It implements two global scans. One is a simple horizontal scan, that has an higher resolution near the horizon. The other scans perpendicular to the horizon line, while starting on the horizon line and scanning away to the bottom.

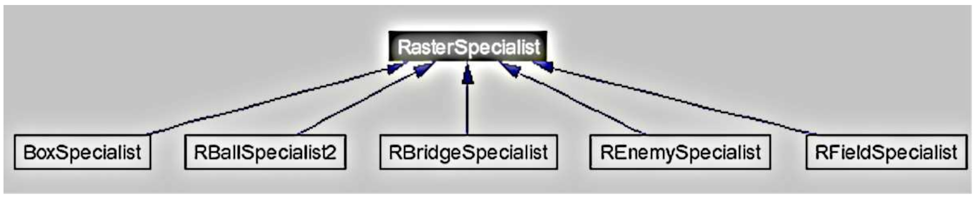


Figure 4.7: Inheritance diagramm for class *RasterSpecialist*.

- The class *BoxSpecialist* implements a beacon and goal detection algorithm.
- The class *RBallSpecialist2* implements the ball detection algorithm.
- The class *RBridgeSpecialist*. This specialist was written for the Open Challenge 2004 and detects the beacons of the platform.

- The class *REnemySpecialist* is an implementation of the opponent detection algorithm.
- The class *RFieldSpecialist* is an implementation of the line detection algorithm.

We have also written some utility classes and libraries that are used by the classes described above, all of them are reusable:

- The class GT2004EdgeDetection implements the algorithms for the edge detection.
- The class RFieldStateMachine imlements a deterministic state machine that classifies edge points on a scanline.
- The file SegmentationTools.h is a library with some collections like lists.

4.3.2 Clustering

Clustering is very important for object recognition purposes, because it is easier to detect an object from a region of pixels rather than from single pixels. We decided to calculate regions with pixels of one or more color classes, dependend on their useage in the various specialists of the RIP. This can be done with the image segmentation algorithm which has been introduced by James Bruce [4]. The algorithm will shortly be described in the following:
Firstly an image is scaned in a grid with parallel scan lines, and store the classification of the several pixels as a subsampled image via *Run Length Encoding* along the scanlines. The scans must be ordered to have the capability to use the second step of this clustering algorithm. If the runs, which are computed from the *RLE*, are sorted in the same way as they had been encoded, we can look on two neighboured scanlines and connect the runs that overlap as shown in 4.8 . To build the regions, we use a *union find* algorithm with path compression [16].

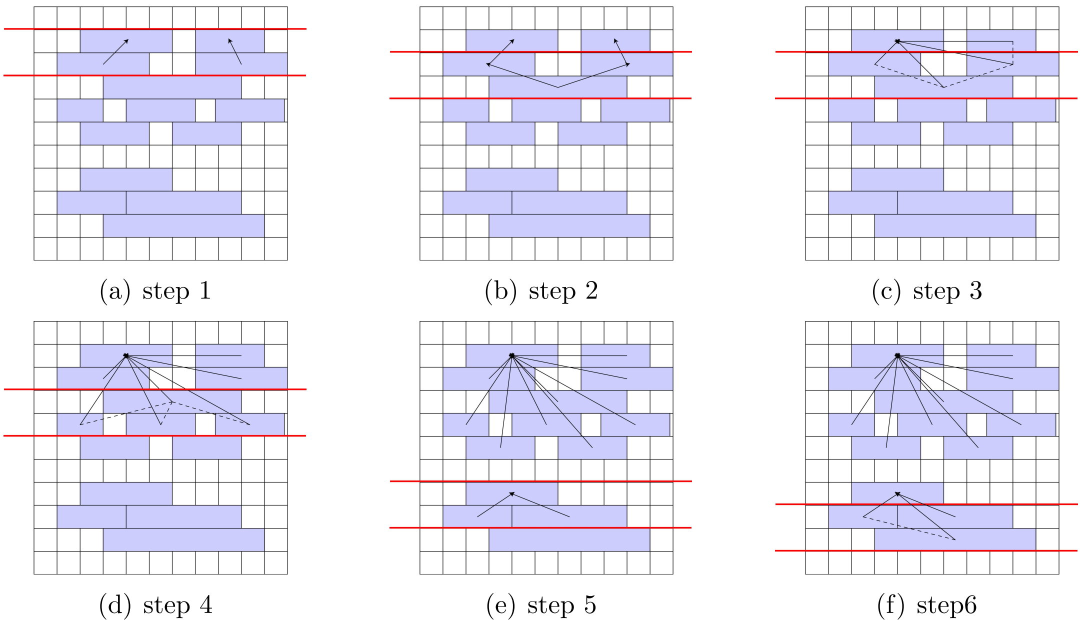

(a) step 1 (b) step 2 (c) step 3 (d) step 4 (e) step 5 (f) step6

Figure 4.8: Illustration of the *region merging* algorithm.

The RLE can be calculated in linear time. The merging of the runs to regions can also be done in almost linear time. A disadvantage of this algorithm is, that it depends on the color classification. If the color classification is bad the clustering of the pixels can be defective. This is a very big problem for clusters of pixels with different color classes, because the color classification of blurred edges is very difficult. For example an edge between pink and yellow can lead to an orange area that splits an object with this two colors in three regions, a pink, orange and yellow one. The pink and yellow regions could not be connected because the orange region that represents the blurred edge, lies between the two other regions.

We tried to handle this problem with a modification of the clustering algoritm described above. The idea was to compare also on further lines than the neighboured ones. The advantage of this approach is that even clusters of pixels with different color classes can be connected quite well. The main disadvantage is that the running time is affected by the number of neighbours we want to compare with every line.

Instead of the *RLE* we used a data structure which we call a Line Pair. A Line Pair consists of two connected points on a scan line covering pixels of one or more colors. On which color they start and on which they end is defined in the scan strategy of the RIP. Basically a Line Pair extends a run of a *RLE* by the additional feature of covering pixels of more than only one color class. Nevertheless it is obvious that the algorithm mentioned above can also be used to cluster Line Pairs as well.

After the *union find* algorithm is done, we usually provide the regions as lists. What could be done in the future is to integrate some additional calculations directly in the clustering routine (e.g. bounding box, color variance, average color, median color, convex hull or centroid).

4.3.3 Ball Detection

The problem of detecting a ball in the image can be reduced to detect orange circles in the image. This is the main idea of the new ball detection implementation. One probleme is that the ball can be partially concealed by other objects. Another difficulty is that a part of a ball could lie beyond the image. So the algorithm for finding circles in the image should be able to complete such circles and interpret them well.

To filter some areas of interest we use all orange clusters found in the area under the horizon line as input data for the detection algorithm. Since only balls are orange on a RoboCup playing field this is a useful and cheap filter for balls. A crucial capability is now to be able to approximate a circle for one of those clusters and generating a validity for the roundness of this cluster. This can be done by a randomized algorithm that needs a collection of edge points $\vec{e} \in E := \{\vec{e_0}, \vec{e_1}, \ldots, \vec{e_k}\}$ as input data to approximate the circle and giving a validity value for it. The authors of GT2003 already implemented a function that calculates a circle from 3 points [5], so obviously we have the capability to calculate a circle from 3 points in constant time. Let δ be the arithmetic average distance of all edge points to the circle. $\vec{m} := m(x, y)$ is the center of the circle and r is its radius. We can calculate δ with:

$$\delta = \frac{\sum_{k=0}^{|E-1|} \left| |\vec{e_k} - \vec{m}| - r \right|}{|E|} \tag{4.5}$$

Now we can formulate the algorithm:

Randomized Circle Fitting Algorithm *(RCFA)*

```
input: edgePoints[],m
output: selectedCircle = null;
variables: bestDist = 10000; dist = 0; circle = null;

repeat m times:
```

1. Select randomized 3 of the edge points and calculate the `circle` that lies on that 3 points.
2. Calculate the arithmetic average distance of the edge points to the `circle` and store it in `dist`.

3.
   ```
   if (dist<bestDist)
   selectedCircle = circle;
   bestDist = dist;
   ```

With the *Randomized Circle Fitting Algorithm* we are able to approximate a circle that has the smallest δ. To detect the needed edge points E for the *RCFA* we implemented a heuristic that uses the edge scanner.

First we calculate the bounding box b of a given cluster C. The center of b is our starting point for the edge scans. We have a selectable number n of scans that go in different directions $\vec{d_k} := d_k(\cos 2\pi\frac{k}{n}, \sin 2\pi\frac{k}{n})$. Now we take the first detected edge point $\vec{e}$ of every scan and put it in E. If a scan reaches the image border we'll continue with the next scan like we find an edge point. By default we make 30 scans for a cluster. After we approximated a circle we calculate its validity. We found two qualities of the circle:

- The **roundness** $\alpha_c \in [0...1]$.
- The **color-pattern-consistency** $\beta_c \in [0...1]$.

α_c can be calculated with a modificated equation out of 4.5 on the preceding page. Let d_k be the distance of a pixel e_k to the circle. We substitute the distance of a edge point to the circle with a new value $w_k := \begin{cases} 1, & if\ d_k <= 2 \\ 0, & otherwise \end{cases}$. And we come to:

$$\alpha_c = \frac{\sum_{k=0}^{|E-1|} w_k}{|E|} \tag{4.6}$$

For the color consistency, we defined a pattern P that says that inside the circle is orange and outside is green white, yellow, red, gray or blue allowed. Now we test a grid G of 10 x 10 pixels $\vec{p_k}$ in the area of the circle, if they meet the conditions of P. Let $c_k := \begin{cases} 1, & if\ \vec{p_k}\ meets\ the\ conditions\ in\ P \\ 0, & otherwise \end{cases}$ be the descision value for every pixel p_i.

$$\beta_c = \frac{\sum_{k=0}^{|G-1|} c_k}{|G|} \tag{4.7}$$

Now we have a aproximation for a circle that can be calculated from a given cluster on a frame and we are able to provide two validity values for the roundness and the color-consistency. We decided to filter balls with the help of the two validity values directly in the image processor. If α_c is less than 0.6 or β_c is less than 0.3, we won't create a BallPercept [5] from the detected circle. A second test is that $(\alpha_c + \beta_c)/2$ is greater than 0.5. These threshold parameters were tuned by hand. This is useful if the color classification is defective and classifies some pixels on other objects as orange. Such a misinterpreted cluster can than be sorted out with the validation tests. Even orange objects that differ only in their shape can be sorted out quite well.

The ball detection algorithm used in the RIP can be described as follows:

Ball Detection Algorithm

1. Sort all clusters by their size.
2. Select the greatest cluster C_m that has not been analyzed. If there is none, STOP.
3. Create some edge points with the scan heuristic described above.
4. Calculate a circle for C_m with a *RCFA* that has a scalable number of iterations to approximate the circle.
5. Validate the circle. If the validity values are high enough follow with step 6, otherwise follow with step 2.
6. Create a BallPercept. STOP.

4.3.4 Beacon Detection

The main difference between the RIP and the GT2003/04 image processor is that the images are mainly horizontally scanned. This choice leads to a completely different approach for the detection of objects, especially for beacons. The beacon detection is effectuated by a specialist called 'BoxSpecialist', whose main characteristic is to create bounding boxes on objects of a given color. Note that this specialist analyses the goals too, since both have a rectangular shape and common colors.

The Beacon detection algorithm as used in the RIP can also be described as in the following:

Beacon detection Algorithm:

1. Find yellow, skyblue and pink Line Pairs in the image in the region of the horizon.
2. Cluster the collected Line Pairs, without considering the colors (see 4.3.2 on page 33).
3. Sort the clusters in decreasing number of Line Pairs.
4. Sum up the length of the Line Pairs of the biggest segments, grouping them by color. The validity test is based on these values.
5. If the proportion of a color is too small compared to the other ones, delete the corresponding Line Pairs.
6. Select the clusters containing two colors, and check if the proportions are similar. The segments that fill all the validity criteria is then considered as a landmark.

As the shape of the obtained cluster is not always rectangular, we need to create a bounding box of it. The horizon has been taken as reference for calculating its corners. This approach has naturally its own limits, since the coordinates of the horizon vector are sometimes inaccurate.
Next, the edge detector (see 4.2 on page 28) is used in order to enlarge the borders of the bounding box, and to fit the real landmark exactly (see 4.3.4). Eventually, a beacon

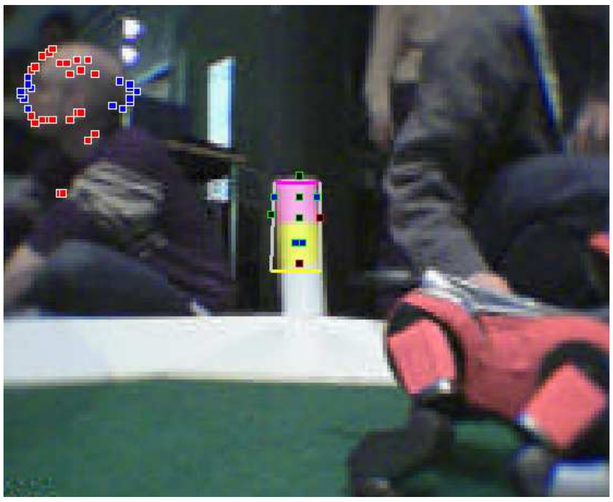


Figure 4.9: A beacon detected by the Box specialist. The points representes edges

percept is generated using the four corners of the resulting bounding box, depending of the color disposition.

4.3.5 Goal Detection

As we explained in the Beacon detection section, the RIP processes both goals and beacons together in a specialist called 'BoxSpecialist', for reasons of efficiency.

The only differences between them are the number of colors and the size. The following goal detection algorithm has also a common part with the beacon detection algorithm:

Goal detection Algorithm:

1. Find all the skyblue and the yellow Line Pairs in the image under the horizon.
2. Cluster them without considering the colors (see 4.3.2 on page 33).
3. Sort the clusters in decreasing number of linepairs
4. Sum the length of the Line Pairs of the biggest clusters, grouping them by color.
5. Remove Line Pairs whose color is negligible compared to the other color.
6. Select the biggests clusters having only one color, and group them by color
7. For both groups: find adjacent clusters if any, and merge them
8. Output the biggest cluster

Note: Step 7 of the algorithm is useful when one robot (usually the goalie) is positionned in the middle of the goal, separating it in two parts or more.

For the same reasons as the beacon detector, the edge scanner enlarges the resulting cluster for a better result (see the red and blue points in 4.10).

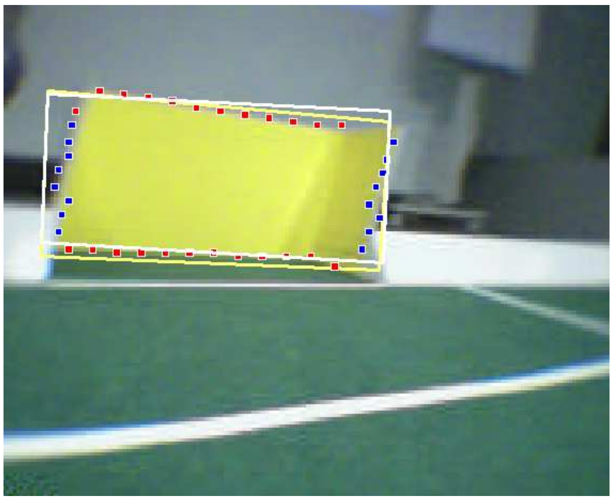


Figure 4.10: A goal detected by the goal specialist. The red and blue points are detected by the edge detector.

4.3.6 Line detection

The lines situated on the field provide precious information for the robot's localization. In some cases, its position could even be determined instantly, without considering the landmarks. A module able to recognize the overall configuration of the field would also be helpful for an efficient localization. The Raster Image Processor contains a specialist called "RFieldSpecialist", who has been designed for the tasks defined above. The idea is to represent the field geometrically using different points:

- The four edges of the areas surrounding the goals
- The intersection of the middle line and the borders
- The kickoff circle (not implemented yet, due to the complexity of the task)
- The corners

These points are estimated by intersecting the lines we find on the field.

We can see a goal corner in the previous picture. A self-locator taking into account this point could be very useful, especially for the goalie, because the localization information is situated much closer than the reference landmarks, giving more precise information.

But the Field Specialist has found another application in the Open Challenge in Lisbon. More exactly, it was used to help the robot to climb the ramp by recognizing the red line situated on it. As the ramp contains only one line, the Field Specialist has been simplified in order to consider only the biggest line it finds. The angular value compared to the horizontal is an indicator for the relative position of the robot (see 4.4 on page 53 for more details).

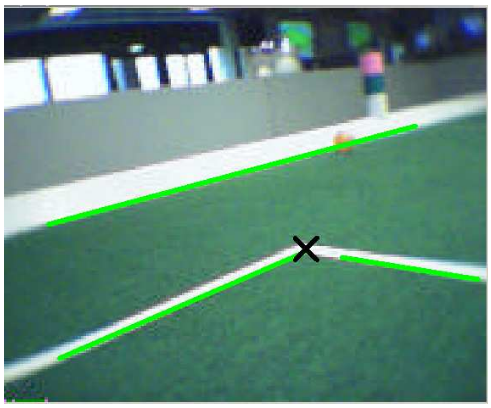


Figure 4.11: The goal area corner has been detected

We will now describe the principles of the Field Specialist.

The scanning method

Basically, the Raster Image Processor uses two scanning algorithms:

- An horizontal one, providing the main information (goals, ball, landmarks, etc) during the prescan phase.
- A scanning method perpendicular to the horizon, providing some important information about the configuration of the field.

The reasons why to choose two scanning methods as opposed to one, like the other image processor does, are the following:

- Scanning an image horizontally can be done very efficiently, as the image points are stored horizontally. A strategy of the RIP, called 'RDefaultStrategy2' includes several optimizations for this task.
- A scanning method is appropriate when a lot of its scan lines can intersect the objects we want to recognize. For this reason, the scan lines have to be perpendicular to the objects. Objects like the ball, the enemies, the goals and the beacons are not strongly oriented, this is the reason why the first scanning method is sufficient to detect them well.
  The red line in the open challenge is vertical most of the time, and detecting its orientation is crucial for climbing the ramp. The horizontal scan is also the only solution. Finally, the lines and the borders of the field are mainly horizontal, this is the reason the second scanning method is used.

The Field Specialist also combines the two scan methods in order to obtain the best results in every situation.
Moreover, the second scan method, effected by a class called 'RFieldStateMachine', is able to differentiate the lines by giving an Id to them (yellow goal, skyBlue goal, field and border).
This information makes the recognition of the intersection of the center line and borders possible.

We will now describe the line recognition algorithm.

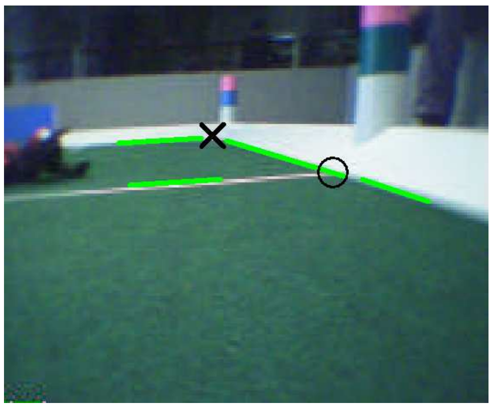
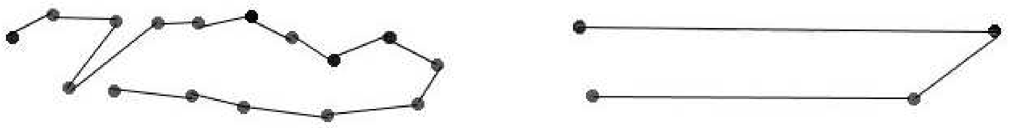


Figure 4.12: Different intersection points on the field

The Line-fitting algorithm

The lines are repesented geometrically (with a starting and an ending point).

```
Input: point[];
Output: line[];
```



(a) the incoming list of points (b) the output lines, not necessary linked

Figure 4.13: The input and output data of the Line-fitting algorithm

1. First, the points need to be sorted.

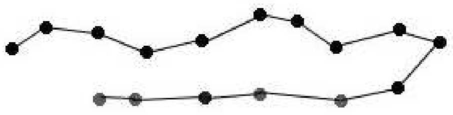


Figure 4.14: the nearest points are linked together

For this task, a specific sorting algorithm has been designed. Before explaining it in detail, we need to define the edges and Line Pair structures used in the Field Specialist. Both derive from a structure called 'Figure', which contains the following attributes:

```
struct Figure {
      virtual int id();
      virtual Vector2<int> ToConsider();
      Figure* next;
}
```

The first attribute is an id: it defines the scan line that was used to find the figure.
The second one is the point to be used in order to calculate the distance between two figures.
The third one is a pointer to the next figure.

Now, we can define the methods formally, and the algorithm itself:
$E := \{f_1, f_2, \ldots, f_n\}$: the set of figures
$C(f)$: the center of a figure.
$I(f)$: the id of a figure.
$s(f)$: the successor of f
$S(f)$: the set of successors starting from f
$G(f) := \{e \in S(f) \mid I(e) = I(s(f))\}$
$H(f) := G(f) \cup \{s(last(G(f))\}$

The sorting algorithm:

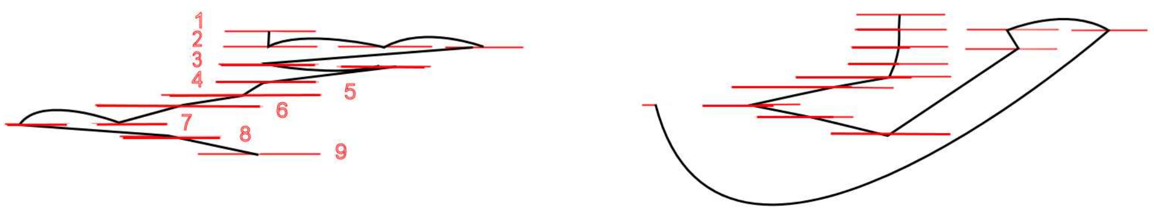
```
Iterate the list: let e be the current element.
```

(a) find $m := \min_{i \in H(e)} dist(C(i), C(e))$
(b) swap $s(e)$ and m

Complexity of the algorithm:
In the practice, the algorithm is quasi linear, as the number of consecutive figures having the same id is very small (maximum 4 or 5 elements)

Here is an illustration of the algorithm:

- Red : The figures (in this case, Line Pairs)
- Black : The pointers of the list
- Numbers: The id of the lines



(a) The input lines (unsorted) (b) The output lines

Figure 4.15: An illustration of the sorting algortithm using Line Pairs

2. For the second part of the Line-fitting algorithm, we choose points sufficiently spaced to form a sub line. If a sequence of sub lines defines equivalent angles, they are merged. If this sequence is big enough, it will appear in the output.

   Here are the most important parameters of the algorithm :

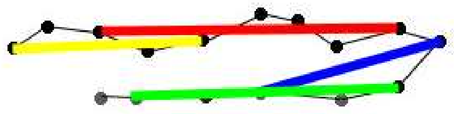


Figure 4.16: A representation of the sub lines

- Step : number of points to jump in order to form a subline
- Alpha : angular tolerance related to the sub lines (in degrees)
- Min size: minimum number of sublines required to form a line

Note that these parameters need to be fine tuned in order to obtain a good approximation, this is the reason why specific parameter sets have to be defined for specific applications. Here are the default parameter set values:

size :*6*, step :*5*, alpha :*20*

The results on the field:

- The pertinence of the result is strongly determined by the quality of the incoming points. As most of them comes from the post scan method of the RIP, a good color table is required to obtain accurate points in a sufficient number.
- The line recogniton is quite good as long as the parameter set is well defined. Too low value of the *alpha* parameter leads to an exaggerated number of lines.
- As many other line detection algorithm, the corners defined by two adjacent lines is not well recognized. This is the reason why output lines are not adjacent. But we can simply intersect them to obtain the desired point.
- The angle values of the sub lines (as defined in 2) can sometimes fluctuate beyond the threshold of the algorithm, although the points are more or less planar. In this case, a final processing phase is needed: distant lines having the same angular value have to be merged (a function has been designed for this task).
- Lines situated beyond half of the ground are poorly detected, since they are too thin to be recognized by the color table (see 4.3.6 on page 40).

We will now talk about the implementation of the line-fitting algorithm. It is based on a library defined in the ImageProcessorTools repository.

The library "SegmentationTools.h"

Here is a description of the tools developed:

1. Slist : template class who differs a bit from the standard implementation. It can naturally be used in other domains than image processing. It was created since the standard list doesn't allows to make special operations like swapping elements, and wasn't very efficient when a lot of elements are stored. The main difference between it and the others lists is that the stored objects contains the pointer to the next element. The slist can also been viewed as an interface that allows to do safe operations on elements pointers.

2. Data structures representing edges and Line Pairs. These structures derive from a class called 'Figure'. The advantage is that edges and Line Pairs can be stored together, wich is very convenient when using several scanning methods, like the raster image processor does. (see the explanation of the Line-fitting algorithm for more details).
   The library contains several data structures for representing a Line Pair (LinePair2, ExtLinePair, etc).

3. Geometrical functions : to compute angle values. The emphasis has been given to calculation speed.Here are the most important ones:

   - *AngleRelativeToHorizontal*(*point* p1, *point* p2) : returns a number between 0 and 360 using trigonometric functions.
   - *theta2*(*point* p1, *point* p2) : same as the precedent function, but approximated. Note that only the first one has been used to compute angle values in the Line-fitting algorithm.

4. Sorting functions: makes the interpretation of a series of figures possible (corresponds to the first phase of the poly-line algorithm).

Note : The majority of the function defined aboved have been designed to work with the slist included in the library.

4.3.7 Obstacle detection

The obstacle detection has to find some free area between the robot and corresponding obstacle points like defined by the obstacle model of the GermanTeam code [5]. To provide data for this model we had to fill the obstacle percept also used by the *GT2003ImageProcessor* [5]. Obstacle points are defined as bottom points of robots, goals, borders or unknown objects on the field.

Since the robots are playing on a green playing field the free area is mostly green, except there is a line between the robot and a obstacle point on the field. Because the obstacle detection of the *GT2003ImageProcessor* has a quite well performance, we decided to implement the same idea of detection algorithm like used in the *GT2003ImageProcessor*. The *GT2003ImageProcessor* scans perpendicular lines to the horizon and decides with a kind of state machine, mainly depending on the color classes of the pixels of the scan line, what kind of obstacle point has been found. The free area in front of the point is also detected.

We reimplemented this idea in the RasterImageProcessor by using the *GT2004EdgeDection* to scan a grid of n scan lines perpendicular to the horizon. The scan lines start on the calculated horizon line, while the direction of the lines points to the area under the horizon line. To get information about the color classes of the pixels between two edge points, we *run length encoded* the color classes appearing on a scan line, while scanning from edge to edge. A state machine called *RFieldStateMachine* is now determing the kind of edge point found in the image due to reading from the color class buffer of the edge detection after it found this edge point. After a scan line has been finished, all edge points found on the scan line are classified and will be provided to the obstacles percept, if they are are classified as obstacle points. Note, that a side effect is the classification of ball points that could be used to detect balls in the image. Even edge points of lines are detected, since the state machine has to seperate them from the border points. Since we classified goal, border **and** line edge points with the help of the RFieldStateMachine, we are also able to fill the *lines percept* used by the *self locator*.

Since the points in the lines and obstacle percepts are represented in the field coordinate system the image processor must perform this transformation. We mainly used a calculation method that depends on the *camera matrix* [5] of the robot. This is not very accurate, because the camera matrix can not be approximated very accurate on a ERS-7 robot. What could be done is to optimize the approximation of the transformation from the image coordinate system to the field coordinate system. This would improve the accuracy of far edge points, that have to be provided to the lines and obstacle percept.

4.3.8 Opponent Detection

Opponent detection based upon the Raster Image Processor is mainly designed for detecting Sony Aibos of type ERS-210. However it is possible to detect Aibos of type ERS-7 with few modifications to the color transitions.
Like in some other specialists of the RIP every frame is scanned horizontally (along the raster) for Line Pairs. The first point indicates the transition from a non-opponent color (green, white, orange, skyblue, yellow, nocolor) to an opponent color (red, blue, gray, black) in the current scanline, while the second point of each Line Pair indicates the opposite transition. (In order to detect Aibos of type ERS-7 it is necessary to consider white as an opponent color.) After a complete horizontal scan of a given frame, all ob-

jects mainly consisting of opponent colors are masked by Line Pairs. Note that only pixels below the horizon are touched since all relevant parts of an Aibo, which are necessary to determine its position are located below the horizon.

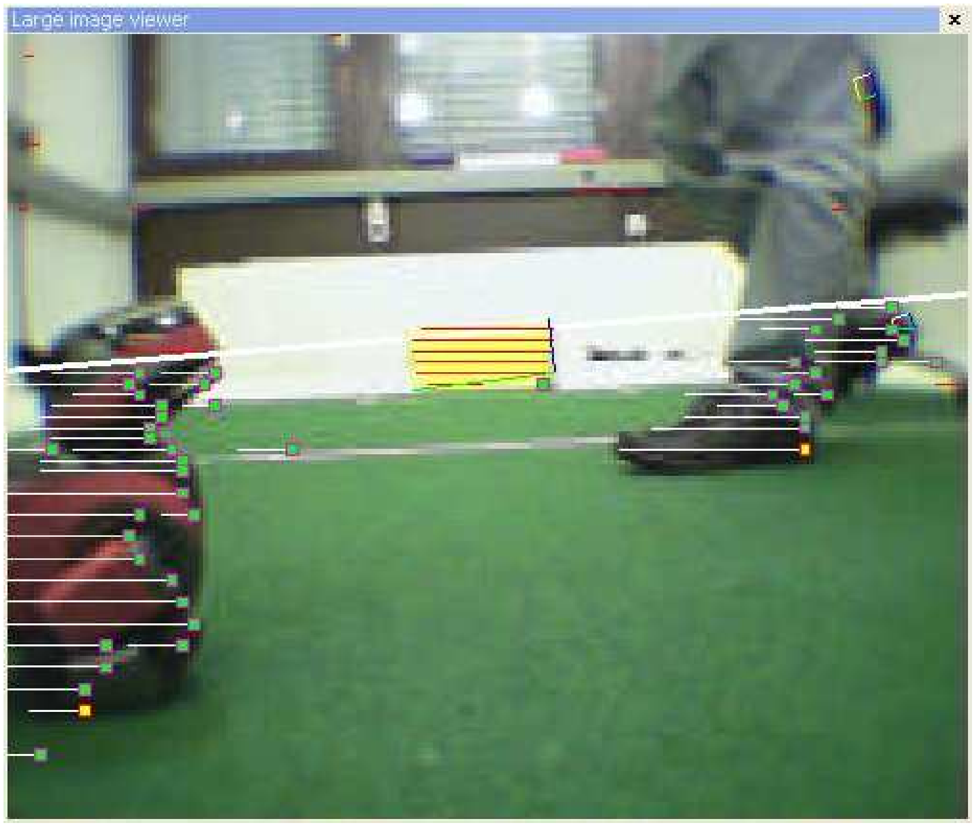
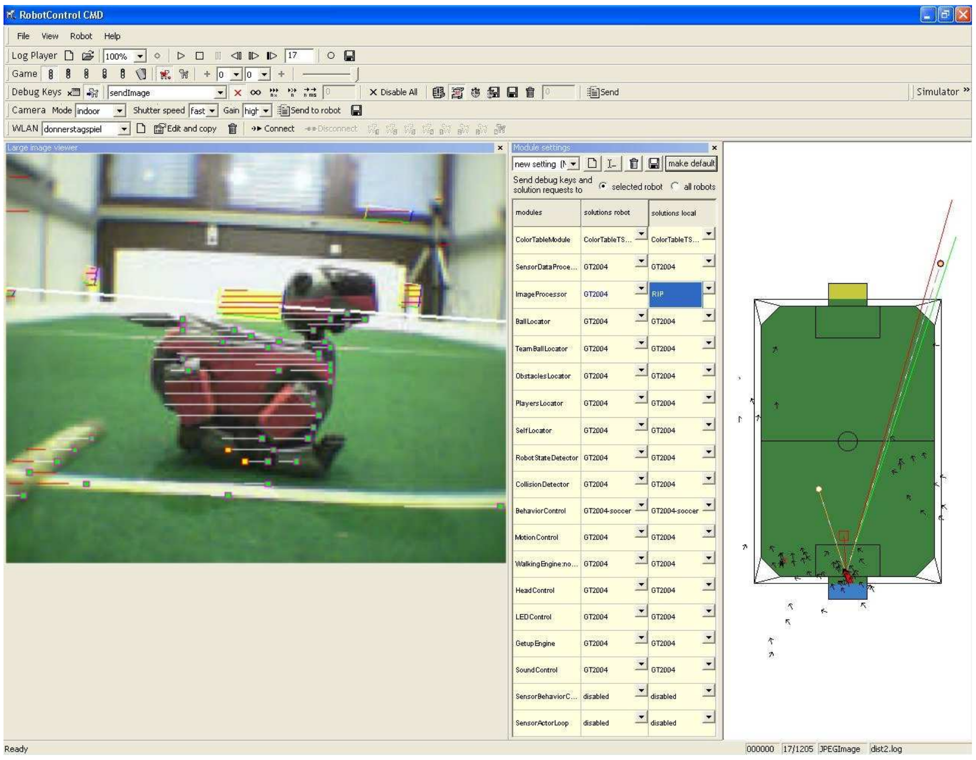


Figure 4.17: On the left a masked Sony Aibo ERS-210 is seen. The thick white line illustrates the horizon, whereas the thin white lines are the Line Pairs. A green point indicates the end of each Line Pair. A yellow point shows which point of the Line Pairs of a given cluster is taken as footpoint. Note that the two Line Pairs near the red Aibo (in its shadow and at the field line) do not belong to the cluster of the detected robot. However they are not recognized as own opponents due to their low validity.

After this first horizontal scan, a postprocessing is needed to differentiate between Line Pairs of different opponents. This is done by a segmentation algorithm, which clusters Line Pairs, that are close together.

At this time we would like to get relative positions of the detected opponents. Therefore a fixed number n of so called footpoints is calculated, which are the points used for mapping to field coordinates. The easiest case is $n = 1$, where for each cluster, the point which is furthest away from the horizon is taken as footpoint. The calculation is accelerated by comparing only those points, which are either on the right side (horizon rises) or on the left side (horizon falls). This point is then mapped to field coordinates relative to the robot, which recognizes the opponent.

For $n \geq 2$ the n farthest points from the horizon are calculated, which are then mapped like in the case $n = 1$ to relative field coordinates. This leads to the question which point to choose as the position of the detected opponent. In this context the Center of Gravity Method is used, which simply means all coordinates of the field points (which are nothing else than 2-dimensional vectors) are summed up and then divided by n. The resulting


RobotControl CMD
File View Robot Help
Log Player
Game
Debug Keys
Camera Mode indoor Shutter speed fast Gain high Send to robot
WLAN donnerstagspiel Edit and copy Connect Disconnect
Large image viewer
Module settings
Simulator
Ready

Send debug keys and solution requests to (• selected robot ○ all robots)

modules	solutions robot	solutions local
ColorTableModule	ColorTableTS...	ColorTableTS...
SensorDataProce...	GT2004	GT2004
ImageProcessor	GT2004	RIP
BallLocator	GT2004	GT2004
TeamBallLocator	GT2004	GT2004
ObstaclesLocator	GT2004	GT2004
PlayersLocator	GT2004	GT2004
SelfLocator	GT2004	GT2004
RobotStateDetector	GT2004	GT2004
CollisionDetector	GT2004	GT2004
BehaviorControl	GT2004-soccer	GT2004-soccer
MotionControl	GT2004	GT2004
WalkingEngine:no...	GT2004	GT2004
HeadControl	GT2004	GT2004
LEDControl	GT2004	GT2004
GetupEngine	GT2004	GT2004
SoundControl	GT2004	GT2004
SensorBehaviorC...	disabled	disabled
SensorActorLoop	disabled	disabled

Figure 4.18: On the left again a picture of a masked Sony Aibo ERS-210, this time using n=2 footpoints. On the right side the red square on the playing field indicates the approximated position of the detected red opponent, which is calculated according to Center of Gravity Method.

vector is a good approximation for the real opponent position and tests have shown that values between 2 and 4 are optimal for n.

Here are the results of the two tests, which were conducted to measure the accuracy of the calculated distance:

DistanceTest 1: The detecting robot has a fixed position in the yellow goal and looks straight to the skyblue goal. The opponent is placed at the line between the recognizing robot and the skyblue goal in a measured distance of 50, 100, 150, 200 and 250 cm, which means the opponent is always seen at an angle of 0°.

Table 4.1: DistanceTest 1 (all distances in cm)

real distance	COG $n = 4$	COG $n = 2$	COG $n = 1$	Frame
50	51,4	49,6	49,4	27
100	115,9	106,8	105,5	322
150	173,4	163,3	161,6	596
200	227,9	211,8	207,7	927
250	328,3	295,9	284,1	1109

DistanceTest 2: The detecting robot has a fixed position in the yellow goal and looks straight to the skyblue goal. The opponent is placed near the left border of the playing field in a measured distance of 50, 100, 150, 200 and 250 cm.

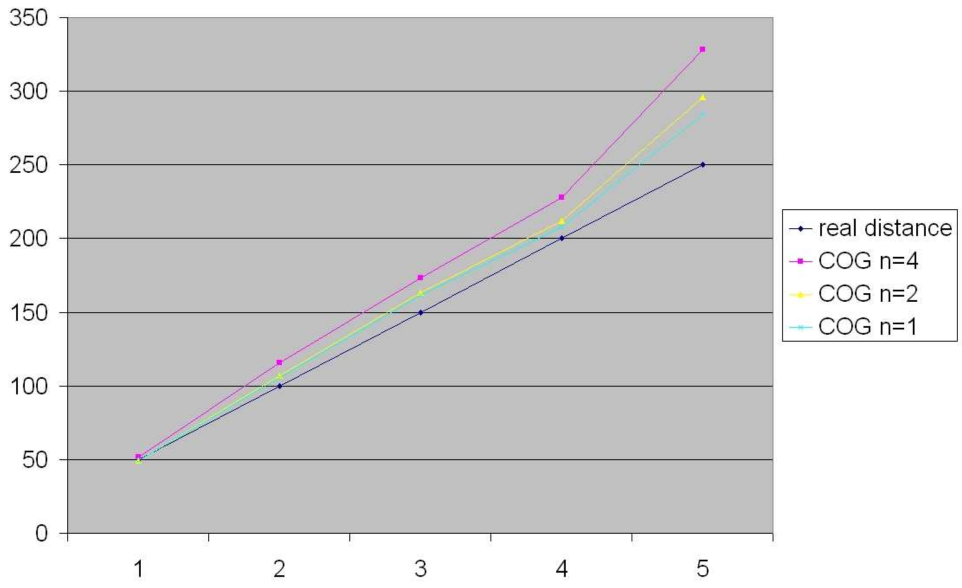


Figure 4.19: The table of DistanceTest 1 as diagram. The closer the calculated lines are to the blue line, which indicates the real, measured distance, the better is the approximation.

Table 4.2: DistanceTest 2 (all distances in cm)

real distance	COG $n = 4$	COG $n = 2$	COG $n = 1$	Frame
50	54,6	53,3	51,4	2
100	107,1	101,7	97	715
150	159,7	150,7	147,1	1240
200	209,3	196,6	184,9	1874
250	N/A	N/A	N/A	N/A

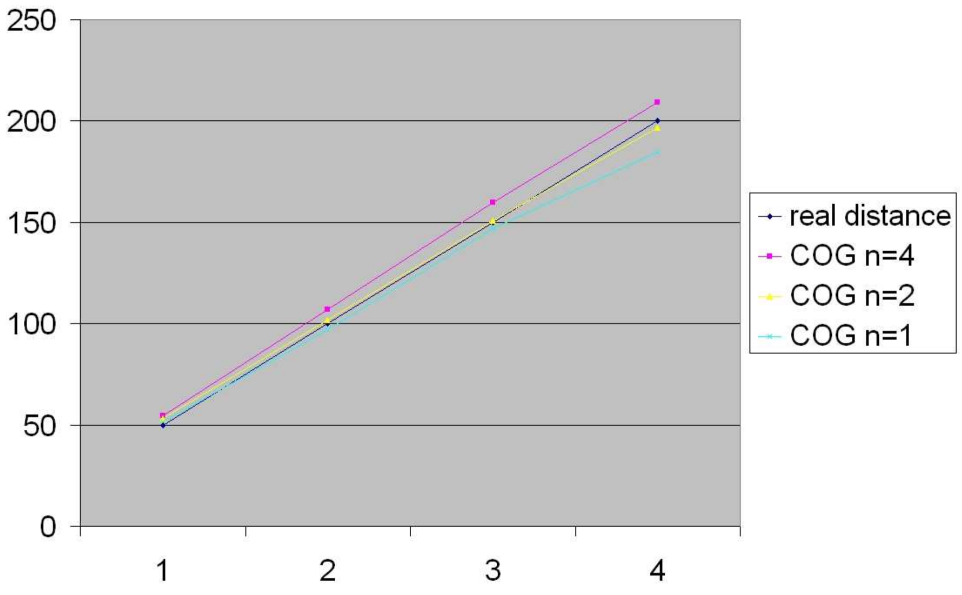


Figure 4.20: The table of DistanceTest 2 as diagram. The closer the calculated lines are to the blue line, which indicates the real, measured distance, the better is the approximation.

Additionally a validity for the recognized clusters is calculated in the *post processing*. Basically the amount of red and blue pixels in the cluster masked by Line Pairs and the calculated distance are taken into account for that. Each cluster is only recognized as an opponent if the calculated validity exceeds a certain threshold.

Problems in the performance

One problem detecting opponent robots mentioned in chapter 4.3.8 on page 44 is the condition of the generated player percepts (compare chapter 2.3.2 on page 9).

A single player percept represents a position of a robot on the field. It is based upon one coordinate which is assumed to be in the middle of the robot. A rectangle surrounds this position.
The problem is that those percepts are generated on n coordinates calculated in the image processing, which are the n farthest points to the horizon of an opponent. In consequence of this a percept does not represent a players position accurately.

Another handicap might be the condition of an above mentioned line pair. A Line Pair starts on every color which apears in a robot, in the current solution red, blue, gray or black. To invoke on gray or black color implicates that all dark areas get considered in the opponent detection, for example shadows or background. Further, opponent segments can get extended by such darker scopes. Resulting disadvantages are ”ghost” opponents (fig. 4.3.8 on the next page and fig. 4.3.8 on the following page), not detected ones (see fig. 4.3.8 on page 50) or consideration of image space, wherein no opponent color can be found(see fig. 4.17 on page 45).
Blue robots, especially of type ERS-210 are harder to detect because of the close distance between blue and black color in the three-dimensional YUV color space (fig. 4.21 on the next page) and a not perfectly adjusted color segmentation as a result of it, which is not unusual dependent on current lightning conditions.

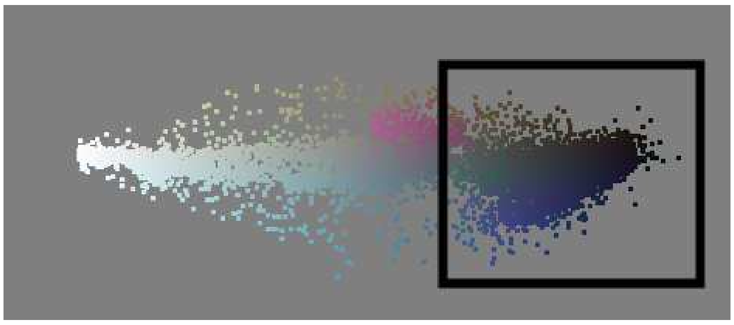
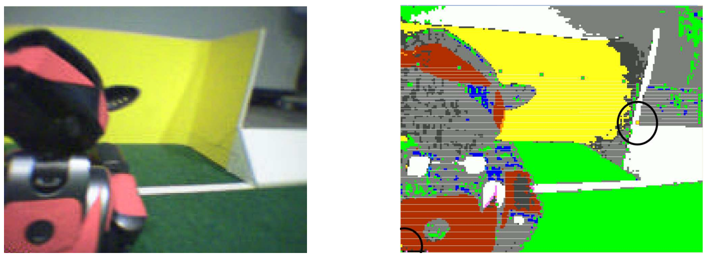


Figure 4.21: appearance of blue and black color in the YUV color space

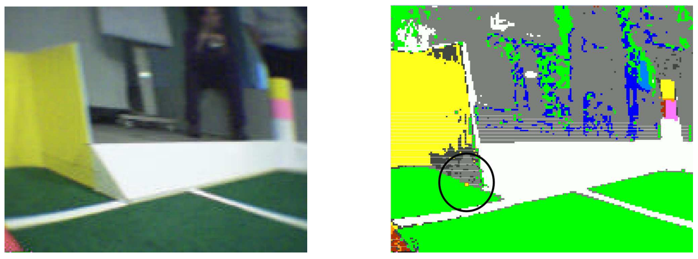
Summary of disadvantages:

1. Currently used percetps calculated on farthest points to the horizon basically do only approximate player positions
2. Close opponents are not considered yet
3. Line Pairs do sometimes cover areas on the image without opponent color in or around them.



(a) Original image (b) Segmented image with debug drawings

Figure 4.22: "Ghost" (marked by the black circle) opponents detected in the background.

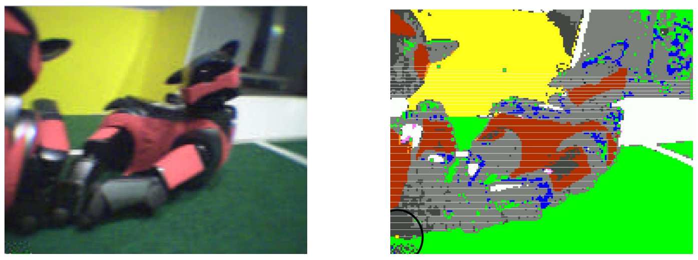


(a) Original image (b) Segmented image with debug drawings

Figure 4.23: "Ghost" (marked by the black circle) opponents detected due to rough color segmentation

4.3.9 Other Approaches to Opponent Detection

Unfortunately the first and the second point mentioned above have not been considered in the following approach for the opponent detection yet. The main difference between



(a) Original image (b) Segmented image with debug drawings

Figure 4.24: Two opponents detected as one. (marked by the black circle)

the first and the alternative solution is the way of collecting Line Pairs and the way to cluster them.
Line Pairs used in this approach represent a consecutive line of pixel and are defined as a 3-Tuple (x_1, x_2, y). The point (x_1, y) marks the starting, the point (x_2, y) the ending point of the Line Pair. On which color they start and on which they end is defined in the strategy of the RIP.

The algorithm distinguishes two kinds of Line Pairs:

- Line Pairs consisting of opponent color only
- Line Pairs consisting of gray color only

The collected Line Pairs have to be clustered to create player percepts. Therefore an alternative cluster algorithm has been created. The basic idea of this algorithm relies in reducing the search space for coherent Line Pairs to their common horizontal position, so the vertical position has not to be considered in this clustering.
An opponent cluster is defined as a 6-Tuple $((x_1, x_2, c, g, d, (x, y))$:

- x_1: Starting horizontal position
- x_2: Ending horizontal position
- c: Amount of opponent colored Line Pairs
- g: Amount of gray colored Line Pairs
- d: Greatest distance to the horizon
- (x, y): Coordinates of the furthest point to the horizon

A 1-dimensional array A[ ] = $\{A[0], \ldots, A[n-1]\}$, n = imagewidth, stores information of horizontal starting and ending positions of all collected Line Pairs to support the algorithm.
Starting positions increment, ending positions decrement the array by an equal value.
By this, the array A[ ] reflects the appearance of Line Pairs related to the horizontal x-axis of an image. Fig. 4.25 on the following page shows a possible situation on the array after a coherent group of Line Pairs would have been collected.
When all Line Pairs are collected, the horizontal starting and ending of opponent cluster have to be found on A[ ] as those interval pairs $[i, j]$, who meet the following equations:

$$S = \sum_{k=i}^{j} A[k] = 0 \quad \wedge \quad A[i], A[j] \neq 0 \quad \wedge \quad 0 \leq i < j < n \tag{4.8}$$

To proof the existence of S in formular eqa:condition, we just need to know that if A[ ] gets incremented on a position k_1 by a value v, it will also be decremented by the same value v on a later position $k_2 > k_1$.

To find all interval pairs, the values of A[ ] are added up beginning from 0 to n. As defined in formular 4.8 an interval starts if S $\neq$ 0 and ends if S = 0 again. Coherent positions of Line Pairs related to the $x-$axis can now be detected by finding the zero points of S.
In fig. 4.26 on the following page, S is displayed after calculation on A[ ] such as shown in fig. 4.25.
By knowing all interval pairs, A[ ] can be transferred to a look up table providing the relevant segment number for a horizontal value x_1 or x_2 (see fig. 4.27 on the following page). Now every Line Pair can be assigned to its cluster it belongs to by looking up the relevant cluster number in A[ ] as $A[x_1]$ or $A[x_2]$. Finally the farthest point to the horizon of each cluster is calculated, which are used to create player percepts.

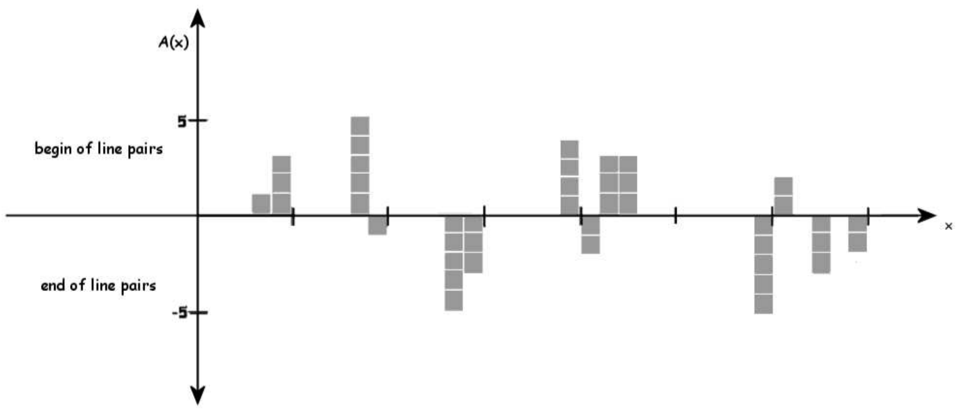


Figure 4.25: Array A[ ] after Line Pairs were collected. The x-axis displays the scan line, the y-axis shows the amount of startings (positive) or endings (negative) of an amount of Line Pairs in the image. Gray squares stand for an increment/decrement, which has taken place.

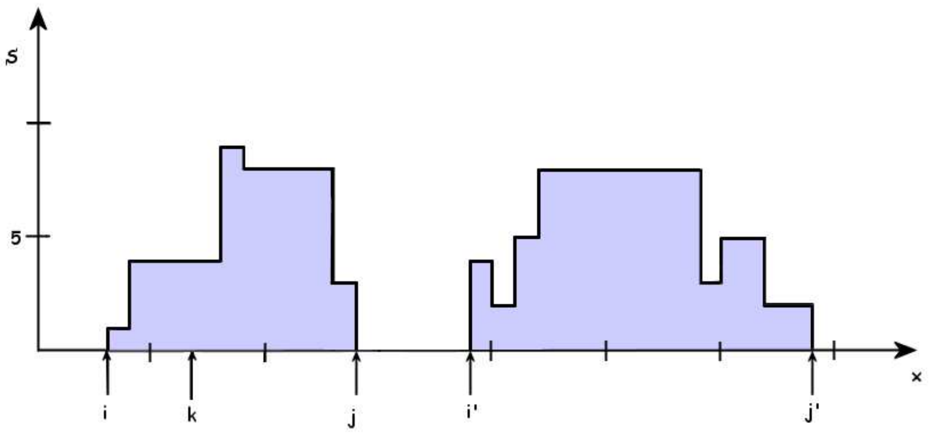


Figure 4.26: the behavior of the sum-function applied on A[ ] beginning from A[i] to A[j]. For example: S(k) = 4.

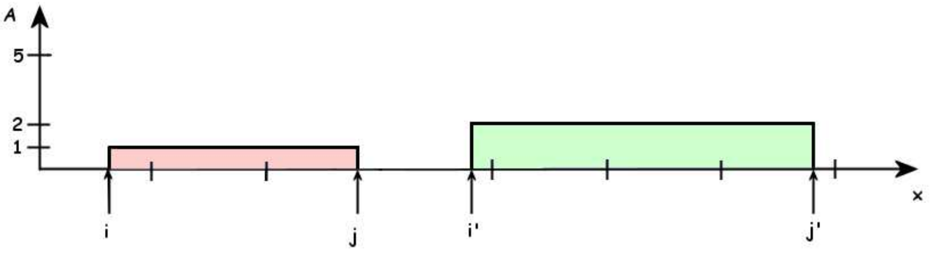


Figure 4.27: Array A[ ] in its final state provides the segment number for each x-value.

In the following the algorithm is described as pseudocode:

Alternative Clustering of Line Pairs:

```
input:  Line Pairs;
output:  opponent segments;
variables:  array A[ ];


during the scan:
```

1. Collect all Line Pairs (x_1, x_2, y) and increment/decrement `A[ ]`:

```
after the scan:
```

1. Find interval pairs `[i,j]` on `A[ ]`;
2. Convert `A[ ]` to a look up table providing the relevant cluster number;
3. Create opponent cluster via `A[ ]`;
4. Create player percetps via furthest point to horizon.

Problems in the performance

Actually the cluster algorithm introduced in this chapter would work properly on a scan parallel to the horizon, which has not been implemented yet. To realize such scan wouldn't be a problem, but within the lack of time at the end of this project group the development has been stopped for now. The algorithm is though not incompatible to a scan parallel to the image, but the bigger the angle is between the horizon and the x-axis of the image, the less precise are the horizontal borders of a cluster. At least for the reasons above mentioned this approach has never been tested on a robot yet.

4.4 Image Processing for the Open Challenge

In the following subsections we will give a brief description for the deployment of the image processing for the Open Challenge. A detailed description of the Open Chanllenge you can find in subsection 7.3.2 on page 83. We had to implement new detection algorithms for climbing the ramp of the platform and finding the feature points of the platform, which the robots had to bite with their mouths. As the image processing solution we used the RIP. For the localization on the field we used a modification of the *GT2003ImageProcessor* as the image processing solution. We switched between the two solutions, because we wanted to get the best performance throughout the open challenge.

4.4.1 Climbing the ramp

In this part of the challenge, we had to face the following difficulties:

- Finding the position situated in front of the ramp
- Turning until the robot faces the ramp
- Climbing the ramp while staying in its middle

The Field specialist of the RIP intervenes during the two last phases of the process, guiding the robot by using the red line.
This problem consists of searching a linear segment of red color, and to determine its angle. The position of the lowest point of the line is a good indicator of the relative position of the robot.

We also need the two following parameters:

α : the angle of the red line (in degrees)
β : the relativ position of the lowest point in the X axis (0.5 corresponds to the middle)

The robot must thus be permanently in such a position that $(\alpha, \beta) = (90, 0.5)$. If not, the parameters $(\alpha', \beta') = (90 - \alpha, 0.5 - \beta)$ are used to correct its trajectory.

4.4.2 Platform Beacon Detection

The four robots, that should move a platform that remotely resembled a *Middle Size League Robot*, must firstly detect some feature points of the platform, which they can bite with their mouths. Two rails at the side of the platform should be the objects, where the

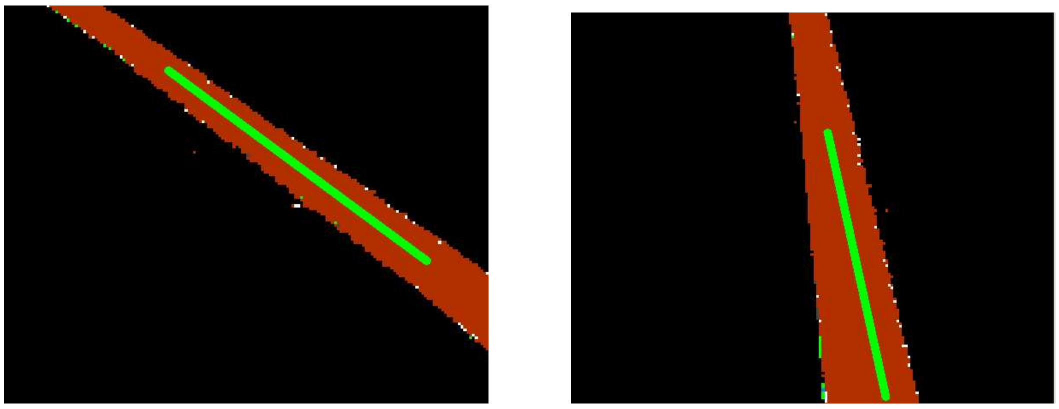

(a) $(\alpha, \beta) = (38, 0.7)$ (b) $(\alpha, \beta) = (73, 0.55)$

Figure 4.28: The output of the red line detector

robots had to bite in. Since the mouth of an ERS-7 robot is not very large, the pipes must be comparatively thin. Another problem was, that the mouth of the robot only will fit to a rail, if the robot stands almost perpendicular to it. So we marked each bite point for the robots with a colored beacon on the pipe and a vertical line centered to the beacon behind it. An illustration of a beacon is shown in 4.29 on the following page. A picture of the original platform we used at the Open Challenge you can find in figure 7.2 on page 84. The idea was that the robot can localize itself relative to the platform, by analyzing the colors of the beacons, the size of the beacons, the direction to the center of the beacon, and the distance of the vertical line to the center of the beacon. So we have to calculate the following variables, that are defined in the field coordinate system :

- p - The position of a beacon, relative to the platform.
- d - The distance of a beacon to the robot.
- α - The angle included by the vector from the beacon center to the robot and the perpendicular to the beacon.
- γ - The angle, where the robot detected the beacon center, relative to the angle of his body.

Every beacon has two colored regions of the same size. Since we gave every beacon an unique color configuration, a robot has some information about the position p of this beacon (see 4.4.2).

left side	right side	position of the bite mark
orange	skyblue	front-left
skyblue	orange	rear-left
yellow	skyblue	front-right
skyblue	yellow	rear-right

Table 4.3: Color configurations of the platform beacons

Since the color configuration of the platform beacons is analogous to the color configuration of the landmarks on the playing field, we used a modification of the detection

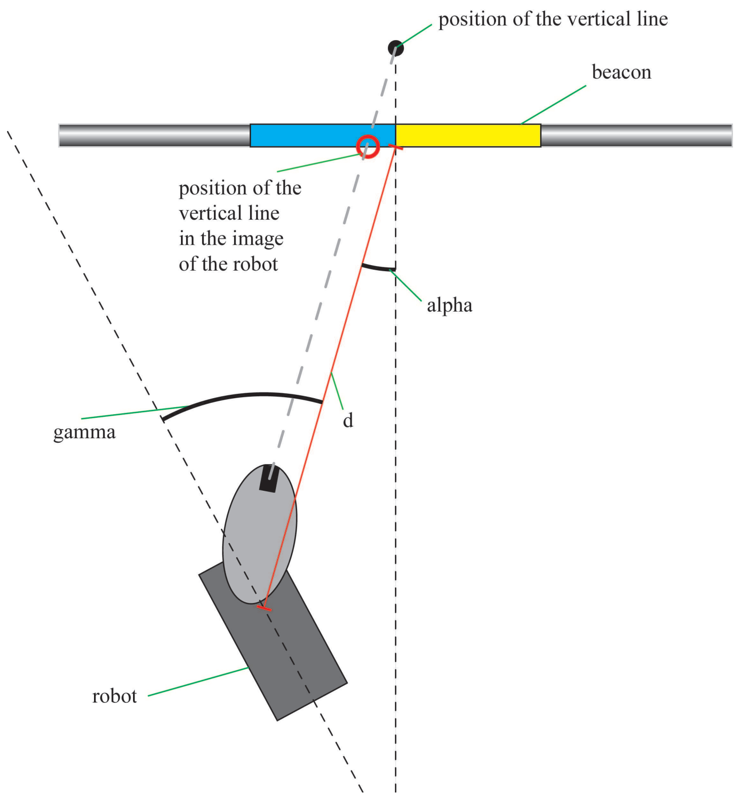

position of the vertical line
beacon
position of the vertical line in the image of the robot
alpha
gamma
d
robot

Figure 4.29: Illustration of a platform beacon.

algorithm explained in 4.3.4 on page 36. The modification was that we searched with the same algorithm for landmarks, which are aligned with the horizon and have the color configuration of the platform beacons. After a beacon has been detected, a scan above the beacon is made, to detect the position of the vertical line behind the beacon.

We calculated the distance d of the beacons by their height. The angle γ can be calculated directly from the position of the beacon center in the image and the *camera matrix*, which represents the position and orientation of the camera. The angle α can be calculated by analyzing the width of the beacon and the distance between the vertical line and the beacon center in the image. We took advantage of the fact, that a robot can know the measurements of the beacons in the real world.

After the variables p ,d, α and γ for a detected beacon have been calculated, the information are provided in a model that considers the odometry data of the robot. All resulting variables are provided to the *XABSL*-Engine as *XABSL*-Input-Symbols [12].

Chapter 5

Resource Scheduling

Creating a *virtual robot* which constitutes as a single interface to a team of soccer playing robots was one of the main aims of the project group. In our approach every single robot of the team represents itself as a coequal actuator to the virtual robot. Thus resource sharing is a substantial part of this approach. As the term *resources* covers a wide range of different meanings we had to examine initially which kind of resources were to be shared and how they were to be implemented into the GT-framework. We distinguished between three kinds or levels of resources: the robot as a whole, processed information and raw hardware like actuators or sensors.
On the level of sharing robots as a whole we developed the *dynamic team tactics (DTT).* This concept allows to define high level team-tasks without the need for caring about allocating jobs to single robots. DTT is described in the next section.
The level of sharing processed information was already existent in the GT-framework when we started with our project group. Since every robot has to process the same kind of data while playing soccer, e.g. detecting the ball, broadcasting the resulting information to the other teammates proves to be the most feasible way. The processed information of the other robots helps to verify the information which was processed locally and is therefore valuable at any time. On this aspect we decided to maintain the given solution of the GT-framework.
To be able to share raw hardware we introduced a new module to the GT-framework, the *resource scheduler module.* It enables a robot to share his own hardware resources and to request remote hardware resources from other robots, e.g. buttons, walking-requests, sound-output, etc. Even though this module was not used in our effort to create the *virtual robot* we wanted to implement solutions for every level of resource sharing postulated by us.

5.1 Dynamic Team Tactics

As aforementioned we developed the *dynamic team tactics* to comply with the needs of sharing robots as a whole. The virtual robot metaphor in mind, we wanted to create a system which enables us to express the behavior of the robot soccer team without caring about the distribution of the different tasks to special robots. The virtual robot should get its assignment and the distribution to his actuators schould be done automatically. Furthermore we wanted the system to be robust against a broken or split network and to be scaleable concerning the number of robots and the number of the available processed

information. Our decision for these requirements was based mainly upon three reasons.
First, as we started with our project group fairly no one knew anything about robot soccer and thus we had to rely on the information given to us by our supervisors and predecessors and the information gained at the workshop in Velbert. Among this, one of the frequently recurring topics was network communication, its importance and its not quite sempiternal stability.
Second, we had a lot of ideas to improve the existing image processing, e.g. detecting enemies(see 4.3.8 on page 44), for which we prepared our resource sharing system. As we did not know if all of our ideas would be accomplished, we designed the system to adapt smoothly to new available information.
Third, the rules (2.1 on page 6) of a robot soccer game contain penalties where the punished robot is taken off the field for 30 seconds. As this can happen in pretty serious situations, e.g. just before kicking a goal or while defending the own penalty area, we wanted the system to adapt instantly by reallocating the tasks to the remaining robots automatically.
In addition to these rather technical requirements we wanted a system which enables us to distribute its development among a variable number of persons as we had a lot of man-power in our project group. Further we wanted to preserve the possibility to use the long lasting and field tested XABSL architecture as basis of our new behavior.

5.1.1 Overview on Dynamic Team Tactics

In our effort to find a viable behavior strategy that fits the above-mentioned criteria, we developed a dynamic system that delegates its tasks, or as we decided to call it: *options*, among the pool of available robots. The system delegates every option to the robot *best suited for it*, it adapts to a *varying number of robots* and always selects a set of options that *fits the overall game situation best.*
An option should be viewed as an atomic action that a robot can perform, such as walking to a specific position, kicking the ball, turning to search the ball and similar things. Every single option is related to a XABSL file which allows us to use existing behaviour features - which means the already existing abilities of the robots as well as the entire XABSL architecture. This partitioning into single options provides the ability to spread the development and especially the testing of the behavior among separated persons or teams. As an example, one team can develop the kicking of the robot and another team can develop pass-playing in parallel. Meanwhile a third team could work on the option ratings (which are introduced below) for those options.
Additionally we created the possibility to group several options in an *option class*, a container for similar options. Assume there is a simple kick option class, then any number of developers can create new kicks and add them independently to this option class without the need of changing the behavior definition. This modularity of our approach makes the development and testing of distributed behavior more fail-safe and suitable for big teams like the GermanTeam or even our project group itself (see 5.1.1 on the following page).
During a game, all robots continuously evaluate how well they can perform every single of those options and produce a *rating* between 0 (very hard to perform) to 100 (very easy to perform). So, for example, a robot that is far away from the ball will assign a very low value for the kickBall option, while a robot that stands next to the ball will assign a high rating. An option rating can use any information it needs which is calculated by the robot. If, for example, an opponent detection is developed, than it is easy to add this new information to the option ratings and do something more sensefull than kicking (maybe

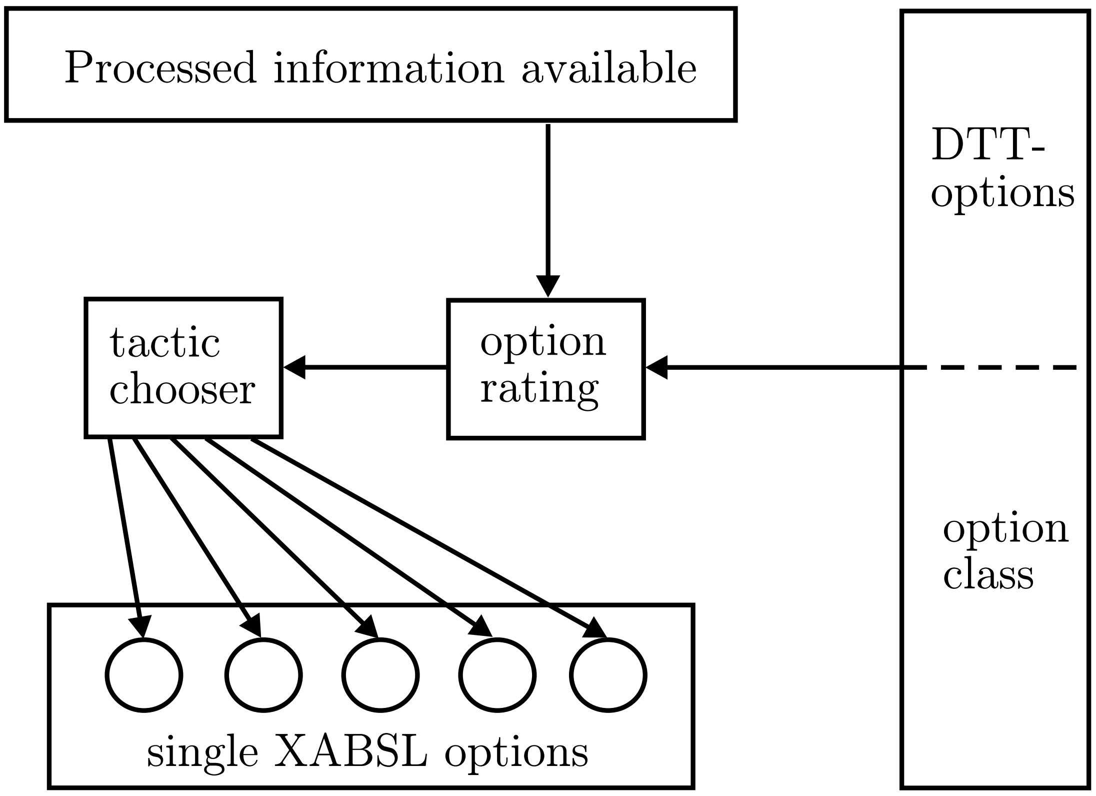


Figure 5.1: Overview of the DTT- Structure

pushing) when you are in front of an opponent. Thus the integration of new processed information is separated of the single behavior options which continues reducing sources of error and gives more flexibility when developing multiple behavior strategies in parallel. All these ratings will be broadcasted so that all robots have all other robot's information as well. This data isn't synchronized but will be ranked down via an *aging algorithm* over time, as it gets outdated. We decided to use broadcasting to comply with our aforesaid requirements. If, for example, a network split occurs, the robots will build up separated teams according to the network breakdown and these separated teams will perform the tasks which they can do best and which are most wanted by the behavior definition.
As an additional layer of control, we have added *global analysers*, that provide the behavior engine with a more general level of information, including information about the game score, offensive or defensive game situations and other global kinds of information. The actual behavior of the robots will be determined in so-called *Tactic Entries* (see 5.1 on the next page). Each of these entries consists of up to n options or option classes, one for every possible robot in the team. Imagine a single tactic entry as a representation of a certain game situation. Each option in a tactic entry can be weighted. This weight represents the importance of this option in such a certain game situation. For example the tactic entry could describe a game situation in front of the opponent goal. There would be certainly an option for kicking the ball, maybe an option for walking to a supporting position, an option for walking to a defending position and an option for staying inside the own goal (*goalie option*). The kick-ball option in this situation should get a higher weight than the go-to-supporting-position option which itself should get a higher weight than the go-to-defend-position option. The weights provide an order among the options of a tactic entry and define thereby how the options are distributed among the robots if the number of robots is less than the number of options in a tactic entry, e.g. through a penalized robot. Besides the weights for each option in a tactic entry, each of the options

NR.	DTT option	weight	allowed robots
0			
1			
2			
⋮			
n			
tactic entry weight			
global analyzer type			
⋮			

Table 5.1: Structure of a tactic entry

can be restricted for some robots. This feature was introduced as the rules 2.1 on page 6 demand a dedicated robot to be the goalie. Thus the other robots of the team should be never advised to execute the *goalie option*.
As the single options in a tactic entry, the tactic entry itself has a weight which represents its importance among the set of tactic entries building the behavior definition. Though these weights are actually fixed values during a game, we envisioned these weights to be actively modified during a game through a learning algorithm which rewards successful tactic entries and punishes the unsuccessful ones.
Additionally each tactic entry has a property list. These properties characterize the tactic entry towards the global analyzers through a set of types provided by each global analyzer. For example the *Offensive/Defensive Analyzer* provides the types *offensive*, *neutral* and *defensive*. Each tactic entry is characterized by one of these three types. During runtime each global analyzer generates a weight for each type which is applied to the tactic entries accordingly. Thus the global analyzers provide a convenient way to create complex and fuzzy sets of tactic entries which are *selected* by global game situation.
To decide which option should be executed, each robot calculates all possible robot/tactic entry combinations and determines a score for each permutation by multiplying the option ratings of the options in the tactic entry with their weights and sum these results. The sum is then multiplied subsequently with the tactic entry weight and the weights generated by the global analyzers. For a team of four robots 24 scores are generated for each tactic entry. Up to 100 tactic entries are a reasonable amount for playing robot soccer so that an overall amount of 2400 scores are calculated. As long as all robots have *reasonably* synchronous ratings, all robots should calculate the same tactic entry with maximum score, choose their assigned option in this entry and perform it. The tactic entries span a decision space which is very smooth and well formed. Even if the option ratings get a little bit asynchronous, the selected tactic entries resemble one another. Even with a total network breakdown each robot will choose the option which it can perform best. This *robustness* is one of the outstanding qualities of DTT.
To give an idea of how this rating works, we will give a short (and simplified) example.
Table 5.2 on the next page shows the option-rating of two robots for the three options *KickToGoal*, *GoToOwnGoal* and *GoToCenter*.
Table 5.3 on the following page shows two tactic entries for two robots including all the weights needed to calculate the final scoring.
Table 5.4 on page 61 shows the calculation of the scores for all Tactic Entries and all robot-permutations. The system will decide to let robot1 kick the ball to the goal and let robot2 go to the own goal, as entry1 for the permutation [robot1, robot2] has the highest score.

	KickToGoal	GoToOwnGoal	GoToCenter
robot1	60	0	75
robot2	0	100	20

Table 5.2: Example for an Ootion-rating of two robots for three options (robot2 is the Goalie, therefore it is only allowed to go into goalie position)

entry1	KickToGoal[10]	GoToOwnGoal[5]	global weight: 2
entry2	GoToCenter[8]	GoToOwnGoal[5]	global weight: 1

Table 5.3: Example for two tactic entries. The corresponding weights are given in brackets

5.1.2 Files, Folders and Implementation

In this part an overview of the used files will be provided as well as a brief description of their function.

Tools/DynamicTeamTactic

The folder *Tools/DynamicTeamTactics* includes all auxiliary and base classes.

- RateableOptions.h/.cpp
  The header file defines the auxiliary RateableOptions class. It includes numerous enums and tool methods that provide information on Options, OptionClasses as well as OptionRating-, TacticChoser- and GlobalAnalyser-Engines.

- CollectedBeliefs.h/.cpp
  These files define the SingleBeliefs and CollectedBeliefs classes. SingleBeliefs is a container for a single robot´s option ratings. CollectedBeliefs embraces all robots´ ratings. Its *update()* method transfers these ratings from the *TeamMessageCollection* to the SingelBelief array. The *broadcast()* method transfers the own ratings into the TeamMessageCollection.

- TacticEntry.h/.cpp
  These files define the TacticEntry and TacticEntryArray classes. TacticEntry includes a single tactic entry, TacticEntryArray contains a number of tactic entries.

- OptionRating.h
  Defines the OptionRating class. Inheriting from BehaviorControlInterfaces, it is the base class for all option rating engines. BehaviorControlInterfaces allows access to all system data, the reference to CollectedBeliefs provides a way to store ratings from the OptionRating-Engine. Also it has a *virtual void rateOptions()* method, that all child classes must implement.

- TacticChooser.h
  Defines the TacticChooser class. Inheriting from BehaviorControlInterfaces, it is the base class for all tactic choser engines. It includes a reference to CollectedBeliefs as well. It has a *virtual RateableOptions::optionsID chooseOption()* method, that all child classes must implement.

- GlobalAnalyser.h
  Defines the GlobalAnalyser class. Inheriting from BehaviorControlInterfaces, it is

permutations	[robot1, robot2]	[robot2, robot1]
entry1	$60 * 10 + 100 * 5 = 1100$ $1100 * 2 = 2200$	$0 * 10 + 0 * 5 = 0$ $0 * 2 = 0$
entry2	$75 * 8 + 100 * 5 = 1100$ $1100 * 1 = 1100$	$20 * 8 + 0 * 5 = 160$ $160 * 1 = 160$

Table 5.4: The calculation of the scores corresponding to the data given in 5.2 on the preceding page and 5.3 on the previous page

the base class for all global analyser engines. Its *virtual void update()* and *virtual double getWeight(RateableOptions::TacticEntryTypeID tacticEntryType)*methods must be implemented by all child classes.

Modules/BehaviorControl/GT2004BehaviorControl/GT2004DTT

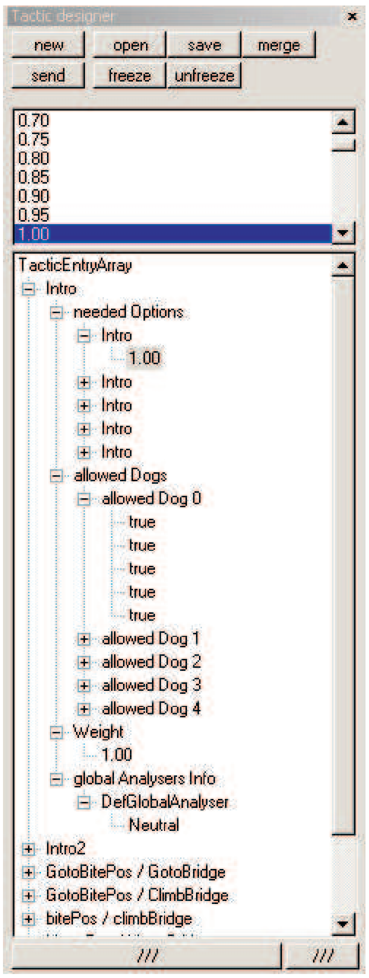
- DefaultOptionRating.h/.cpp
  DefaultOptionRating implements our current version of the option ratings.
- DefaultTacticChooser.h/.cpp
  In this file we implemented a tactic chooser for the DTT system.
- OffDefAnalyser.h/.cpp
  This file provides the offensive/defensive global analyzer currently used.
- GoalieCoach.h/.cpp
  This file provides the goalie coach global analyzer currently used.

5.1.3 Working with DTT

After we developed this basic behavior-engine, we started to implement it within the GermanTeam framework. Since there are only 4 robots in one team, we decided to brute-force all 24 possible robot-option combinations instead of developing more sophisticated algorithms that might have polynomial calculation time in the number of teammembers. Therefore the engine was quite easy to implement.
As above-mentioned, DTT allowed us to spread the behavior work over different subgroups of our project group where these subgroups were able to develop their tasks quite independently from each other.

Figure 5.2 on the following page shows the GT2004 RobotControl dialog called *tactic designer* that is provided by the DTT-engine to edit and add new tactic-entries. This tactic entries are saved into a file named *default.dtt* and copied to the stick[1]. The tactic designer displays the set of tactic entries in a tree structure which can be edited directly with a click-wait-click scheme or by selecting an appropriate value in a list box placed above the tree view. The list box changes its content interactively depending on the type of the selected item in the tree view.
Since it's quite hard to directly observe the reason for a misbehavior during a running game on the playing field, we implemented a real-time option-rating viewer (see figure 5.3 on page 63) in the framework. It displays the current ratings for all options and the tactic-entry every robot has chosen. By using these tools, we were able to start the work

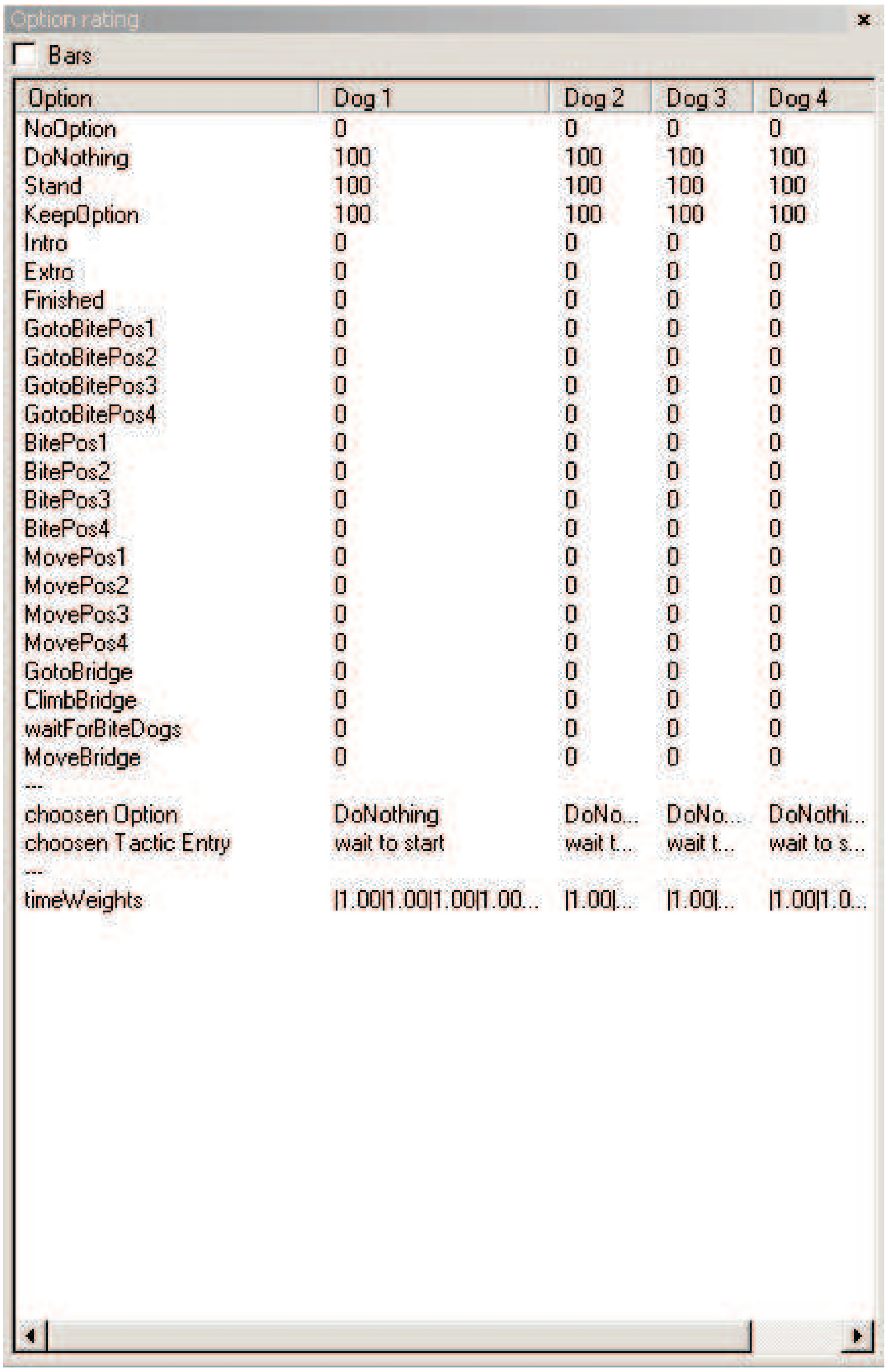
[1]*stick*: short for *Sony Memory Stick* - a removable memory storage chip used by the Aibo robots



```
0.70
0.75
0.80
0.85
0.90
0.95
1.00
```

```
TacticEntryArray
⊟ Intro
   ⊟ needed Options
      ⊟ Intro
         1.00
      ⊞ Intro
      ⊞ Intro
      ⊞ Intro
      ⊞ Intro
   ⊟ allowed Dogs
      ⊟ allowed Dog 0
         true
         true
         true
         true
         true
      ⊞ allowed Dog 1
      ⊞ allowed Dog 2
      ⊞ allowed Dog 3
      ⊞ allowed Dog 4
   ⊟ Weight
      1.00
   ⊟ global Analysers Info
      ⊟ DefGlobalAnalyser
         Neutral
⊞ Intro2
⊞ GotoBitePos / GotoBridge
⊞ GotoBitePos / ClimbBridge
⊞ bitePos / climbBridge
```

Figure 5.2: A screen shot of the tactic-entry editing window. The entries currently loaded define the behavior for the open challenge (see 7.3.2 on page 83).



Option	Dog 1	Dog 2	Dog 3	Dog 4
NoOption	0	0	0	0
DoNothing	100	100	100	100
Stand	100	100	100	100
KeepOption	100	100	100	100
Intro	0	0	0	0
Extro	0	0	0	0
Finished	0	0	0	0
GotoBitePos1	0	0	0	0
GotoBitePos2	0	0	0	0
GotoBitePos3	0	0	0	0
GotoBitePos4	0	0	0	0
BitePos1	0	0	0	0
BitePos2	0	0	0	0
BitePos3	0	0	0	0
BitePos4	0	0	0	0
MovePos1	0	0	0	0
MovePos2	0	0	0	0
MovePos3	0	0	0	0
MovePos4	0	0	0	0
GotoBridge	0	0	0	0
ClimbBridge	0	0	0	0
waitForBiteDogs	0	0	0	0
MoveBridge	0	0	0	0
---				
choosen Option	DoNothing	DoNo...	DoNo...	DoNothi...
choosen Tactic Entry	wait to start	wait t...	wait t...	wait to s...
---				
timeWeights	|1.00|1.00|1.00|1.00...	|1.00|...	|1.00|...	|1.00|1.0...

Figure 5.3: This window shows all the current option-ratings of the four robots in real-time. The visible options are those needed for the open challenge (see 7.3.2 on page 83).

on the tactical behavior of our robot soccer team. After a few attempts, we found out that the global analyzers are a very useful and strong tool to organize subsets of tactic entries. The first global analyzer we implemented was the offense/defense analyzer, which made it a lot easier to create an offensive and a defensive focus in the strategy. In fact, it's quite simple to decide if the team is in the offense or the defense just by having a look at the ball-position. If the ball is in the own half, the team has to play defensively, offensively otherwise.

Based on this engine, we set up a game against the GermanTeam code from Padova. In that game DTT was resonable for the positioning of the robots and the old behavior for the other actions. Both teams were ERS-210 and had the same color table, so the only difference was the behavior. We had several testing games with this setup and the DTT based team won 4 out of 5 games. After these encouraging results, we decided to migrate more of the old behavior solutions to DTT. To do so, we had to focus especially on the option ratings, because they are the core part of the entire system.

5.1.4 The inner workings of option ratings

An option rating is basically a relation, that is assigning numbers between 0 and 100 to different options based on the current sensor data and the decisions the team has made before. The GoToBall rating is a good example describing how an easy option rating works.
Our first approach for the GoToBall rating was a simple, linear formula:

$$max(0, min(100, -0.1 * d + 100))$$

The min/max part of the formula simply ensures that all ratings are within the range of 0 to 100, the real rating is done by 0.1*d+100, where d is the distance of the ball to the robot in millimeters. If the distance is 0, the rating is 100. The rating will fall down linearly with d and reach its minimum (0) at a distance of 1000 mm as can be seen in figure 5.4 on the following page.

On the one hand, these linear ratings are very easy to calculate, but on the other hand, they are quite hard to handle in situations that allow different tactics. In those cases, linear ratings suffer from the noisy input data of a robot and can lead to change the chosen tactic every other frame which means a behavior that is based on linear option ratings is not stable.
The first tuning approach to the linear option ratings was to use multiple input variables instead of only one. In the example of the GoToBall rating, we added the angle to the ball to the rating (if a robot looks straight to the ball, it has a better GoToBall rating than a robot that has to rotate to be able to approach the ball). These changes lead to a better behavior when the ball was positioned right between two robots. Another change we intended to incorporate was to take the positions of the opponents into account. Unfortunately this was not possible, because the behavior module was never supplied with information about the opponent´s positions (see 4.3.8 on page 44 for details).

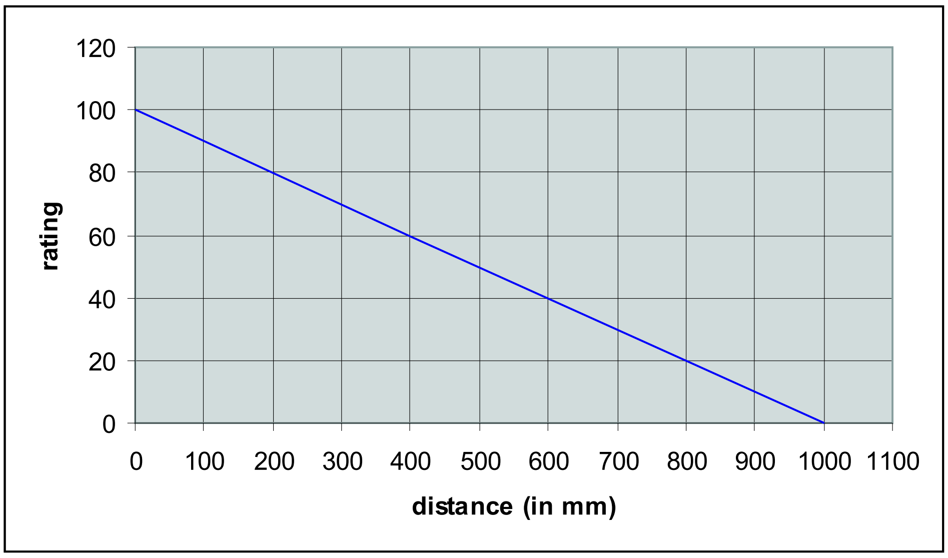


Figure 5.4: Example of a very simple, linear option rating for GoToBall

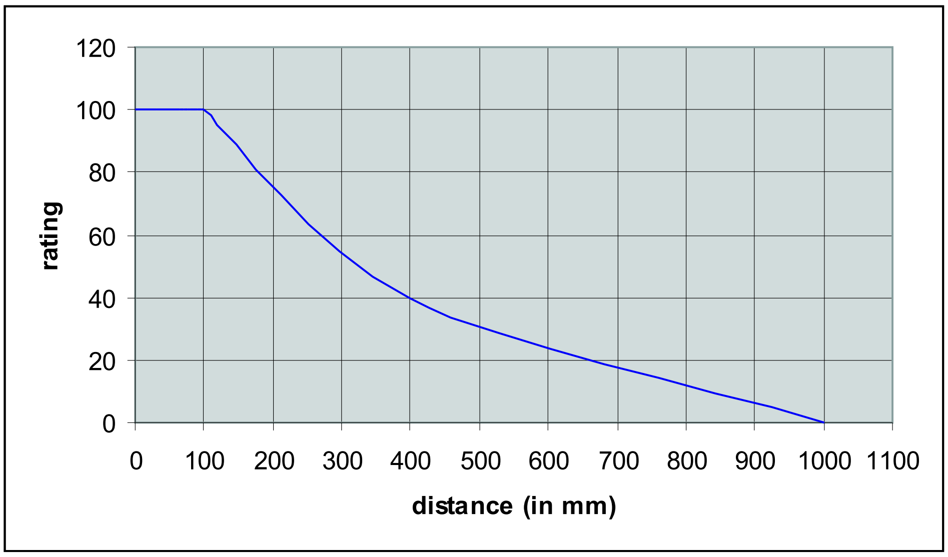


Figure 5.5: The 'theoretical optimal' option rating for GoToBall

Because polynomial functions are not easy to tune, we used quadratic or cubic ratings only very seldom. Instead we combined different linear ratings into one. This led to a linear approximation of the desired rating in different value-segments. The figures 5.5 and 5.6 on the next page show this technique in greater detail.

Apart from these continuous ratings, we used even simpler ones (i.e: when ball in range from 0 to 50 mm: rate to 100, for greater distances: rate to 0). These were very useful for the reflexing goalie we implemented based on DTT, because the reflexing is only triggered if the following four conditions are fulfilled:

1. ball is closer than a specified distance
2. ball´s speed exceeds specified threshold
3. ball´s moving direction is inside specified angle range

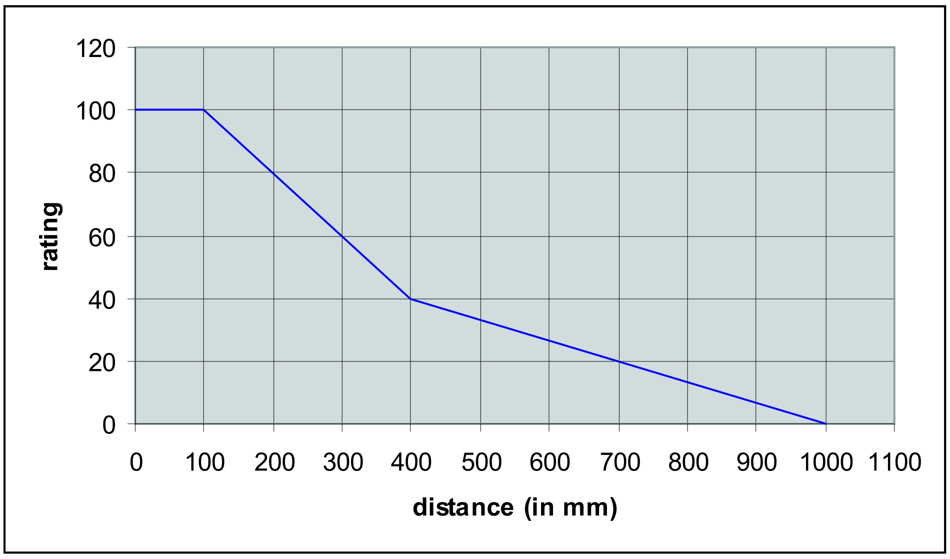


Figure 5.6: A linear approximation of the rating shown in 5.5 on the preceding page.

4. time since last own reflex movement above a specified threshold

5.2 Scheduler Module Integration

The development and implementation of the resource scheduler module completes the integration of our resource sharing task into the GT-framework. With this module it is possible to share the hardware of a robot on a very low level.
We made the experience that this feature is not constantly used while playing soccer. Since every robot on the soccerfield faces the same problems, e.g. detecting the ball, the load on every single robot is typically similar. Thus information that may be useful and could be extracted from the sensor data of another robot is normally calculated by this latter robot anyway. Directly sharing hardware of a robot on such a low level is more suitable for debugging purposes. In that case it is possible to use button and sensor input, LEDs and sound output of a robot to remote control and monitor another robot. As this case is still not so frequent, we decided to implement this kind of resource sharing in a *request and grant* fashion. For that reason the teammessage structure was expanded to notify the other robots about the resources a single robot wants to share, to request a resource of another robot and to send a return value to the requesting robot. To notify the other robots about the resources a robot is willing to share, the *outgoingSharedResourcesTeamMessage* was added to the *TeamMessageCollection*. For requesting a shared resource the *outgoingRequestResourceTeamMessage* array was added to the *TeamMessageCollection*. For each teammate the array provides an entry. To return requested data and sending acknoledgements to the requesting robots the *outgoingReturnResourceTeamMessage* array was added to the *TeamMessage-collection*. Like the *outgoingRequestResourceTeamMessage* the array provides an entry for each teammate. As an example assume the following situation: robot A wants to share its sensor information stored in the sensor data buffer. To do that, it just sets the *isShared[rtSensorDataBuffer]* member of the outgoingSharedResourcesTeamMessage to *true*. This message is automatically transferred to all other robots. One of these robots, e.g. robot B, wants to request the sensor data from robot A. Having just received the *SharedResourcesTeamMessage* from robot A it knows that

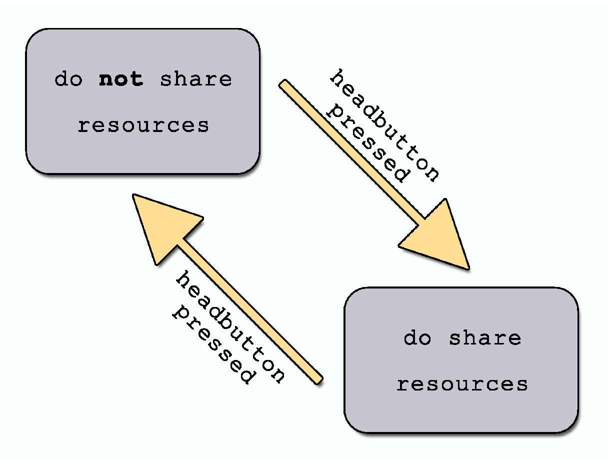


Figure 5.7: state machine of senso *slave*

robot A is willing to share the resource it wants to request. By setting the type member of the *outgoingRequestResourceTeamMessage[A]* to *rtSensorDataBuffer*, robot B request this sensor data buffer resource from robot A. Once this Message is received by robot A, robot A sends its current sensor data buffer back to robot B via the *outgoingReturnResourceTeamMessage[B]*.
As we thought about using the resource scheduler module mainly for providing debug assistance, our solution for the scheduler module uses the *first come, first served* approach. Due to the modular concept of the GT-framework it is possible to implement other scheduling strategies if required in a plain way as new solutions for this module.

5.2.1 Senso - a sample application

To demonstrate the mode of operation of the resource scheduler module we implemented a distributed version of the game *senso*[2].
Senso works as follows: the computer plays a sequence of light and sound signals and the human player has to repeat this sequence by pressing the appropriate buttons. By increasing the number of light and sound signals in the sequence each level, the degree of difficulty increases continually.
In our implementation of senso, one robot uses the LEDs, buttons and the sound-output of a variable number[3] of other robots to play the sequence of light and sound signals and to get information about the correct repetition of this sequence. The state machine of the *slave* robots is expectedly simple (see figure 5.7). At the pushing of the headbutton the robot switches into a state where it shares its LED, sound and button resources. As aforementioned the main robot recognizes the shared resources of the other robots and generates a corresponding light and sound sequence. Afterwards this sequence is played on the other robots and the main robot waits for the correct input of the player. If the player fails, the game stops and a new game can be started by pressing the back button of the main robot. If the player succeeds, the sequence of the next level is generated and played. At the beginning of each level the main robot checks for new robots willing to share their resources and robots which deny sharing their resources respectively. Thus the number of robots which participate in the game can alter from level to level completly seamlessy.

[2]Also known as *Simon Says*.
[3]Currently up to three other robots are supported.

5.2.2 Conclusion

The senso example makes, unlike debugging, heavy use of the resource scheduler module. Under these conditions we encountered additional need of network bandwidth caused by the request and grant scheme of the resource scheduler module even though the used scheme is quite simple structured. In contrast to one packet each 500ms used by the aforementioned dynamic team tactics we had to increase the sending rate to one packet each 75ms to establish a *fluid* game. As the actual GT-framework limits the communication speed to one packet per frame of the cognition process (which runs typically between 20 to 25 fps), especially sending pictures comes along with big delays. In this regard we evaluated the operating expense of modifying the GT-framework to minimize those delays and came to the result, that it would be a disproportional effort needed to do so.
We assess this as a further indicator that resource scheduling on such a low level is not as suitable for a complex task like playing soccer as approaches like dynamic team tactics which are situated on a significantly higher level.

Chapter 6

Ceiling Camera

One of the tasks of our project group was to develop a model for automatic situation analysis. As described in the previous chapters, many different methods, like opponent detection, Dynamic Team Tactics, automatic learning of new kicks etc. were tried and implemented on the robot. But all these methods suffer from one common problem: noisy and imprecise robot hardware. While there can't be anything done against this problem in the actual game, a higher precision localization, ball detection and enemy recognition is a crucial feature for most automated learning tasks.

One example is the current method of automated kick learning. The robot has to localize itself on the field, find the ball, kick it, and try to estimate the ball's new position after the kick. If done on the robot hardware alone, all these measurements are noisy, and have accuracies that are in the 10s of centimeters range. Also localizing and finding the ball takes a lot of time, and repeated kicking stresses the robot hardware.

Other problems were due to the introduction of the new ERS-7 model. The self-locator was originally developed and tuned for the ERS-210, and performed badly on the new robot model. While the original authors had used a laser scanner mounted at the side of the field to gain ground truth about the robots' position [5], we had no such technologies at our hand. Without ground truth, tuning the self locator turned into a matter of guesswork and drawing conclusions from the robots' behavior.

To ease the situation, a plan for implementing a ceiling mounted camera was devised. The camera should be able to capture the complete playing field, and enable us to directly measure positions, distances and velocities on the field.

The development plan consisted of two parts:

- Implement a realtime, interactive ceiling camera view, that can be shown in the RobotControl field view. To directly evaluate the robot's performance it should be possible to overlay the robots' debug drawings, to see both ground truth and debug data in direct correspondence.
- Implement a ceiling camera oracle that can automatically recognize and measure the robots' and ball's position on the playing field. A network server should be developed, so that the oracle has to run on only one computer in the arena. This oracle could also be used for various other tasks like automatic evolution of new behavior, testing the obstacle and opponent detection code and learning of new kicks.

The two part plan seemed to be the best way to implement the desired tools, because it divides the task into smaller, not-too-difficult development steps. As we will see, the second step is directly based on the first one, and can strongly benefit from code reuse.

Using a camera for precise measurement of distances isn't as easy as one might think at first. Camera pictures mainly suffer from two problems: lens distortion (because lenses aren't perfect) and perspective distortion (because the camera sometimes can't be mounted exactly above the middle of the field). The two are also known as the *intrinsic* and *extrinsic* parameters of a camera setup.

6.1 Lens distortion correction

After evaluating different camera models, it became clear that a wide-angle lens is needed for the given size of the playing field and the limited mounting height. By using a wide-angle lens, it is possible to capture the entire playing field with only one camera mounted above the center of the field. As a downside, wide-angle lenses often exhibit pronounced lens distortion (also known as *fish-eye effect*), that has to be corrected before the picture can be used for measurements. An example of this fish-eye effect can be seen in Figure 6.1(a) on the following page.

The topic of distortion correction for camera lenses has been widely researched since the early 1950s because of its importance for aerial photometry. Clarke and Fryer [7] give a good overview of the history and theory of distortion correction. In the last years, many methods were developed for automatic and semi-automatic estimation of lens parameters; after evaluating some of the methods, we chose the lens distortion model presented on J. Bouguet's website [1] (which is based on a popular paper on lens distortion correction written by D.C. Brown [3]).

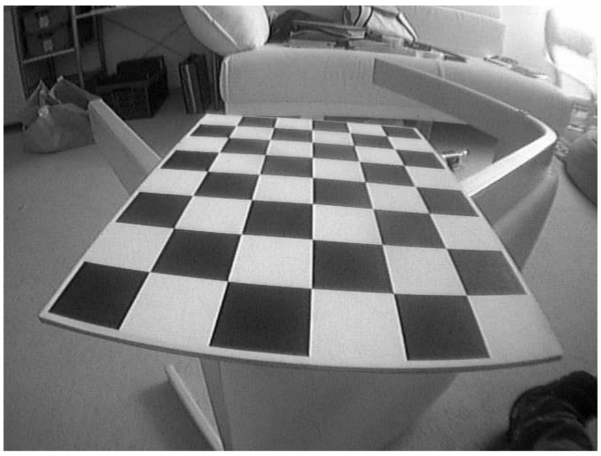
The reasons were mainly ease of use – there's a complete calibration toolkit for MATLAB written by the authors of the website, and the calibration can be done semi-automatically by taking shots of a checkerboard pattern (see Figure 6.1(a) on the next page) under different angles. According to the authors, subpixel accurate distortion correction is achievable without problems.

The intrinsic distortion model used by our software has the following parameters:

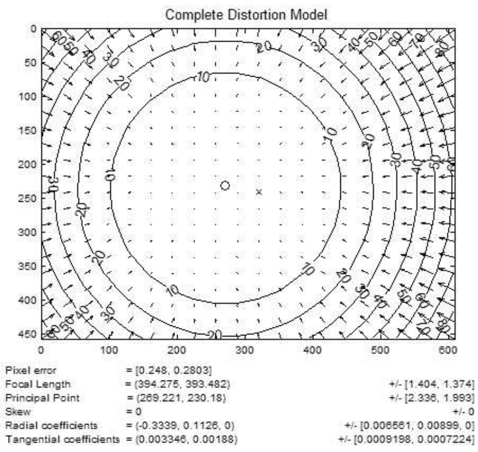
- **Focal length:** The focal length in pixels stored in (f_x, f_y)
- **Principal point:** The principal point coordinates (c_x, c_y)
- **Skew coefficient:** Angle between x and y axis, α
- **Distortions:** Radial and tangential coefficients, $(k_1, \ldots, k_5)$

Let (x, y) be the coordinates of a pixel in the original camera image. Let

$$\begin{aligned} x' &= \frac{x - c_x}{f_x} \\ y' &= \frac{y - c_y}{f_y} \\ r &= \sqrt{x'^2 + y'^2} \end{aligned} \tag{6.1}$$



(a) Calibration object - note that the checkerboard's edge appears bent though it is perfectly straight in reality



```
Pixel error              = [0.248, 0.2803]
Focal Length             = (394.275, 393.482)            +/- [1.404, 1.374]
Principal Point          = (269.221, 230.18)             +/- [2.336, 1.993]
Skew                     = 0                             +/- 0
Radial coefficients      = (-0.3339, 0.1126, 0)          +/- [0.006561, 0.00899, 0]
Tangential coefficients  = (0.003346, 0.00188)           +/- [0.0009198, 0.0007224]
```

(b) Intrinsic parameters estimated by the MATLAB toolkit. The small arrows show the pixel displacements due to the lens distortion.

Figure 6.1: Lens distortion correction

Then x', y' are the normalized image coordinates. We can now write down the distortion model as a set of three correction terms:

- **Radial distortion**

$$\begin{aligned}\Delta r_x &= (k_1 r^2 + k_2 r^4 + k_5 r^6)x' \\ \Delta r_y &= (k_1 r^2 + k_2 r^4 + k_5 r^6)y' \qquad (6.2)\end{aligned}$$

  k_1, k_2, k_5 are also known as the $2nd$, $4th$ and $6th$ order *radial distortion coefficients.* $(\Delta r_x, \Delta r_y)$ is the offset from the optimal pixel for a given distance from the center of projection.

- **Tangential distortion**

$$\begin{aligned}\Delta t_x &= 2k_3 x'y' + k_4(r^2 + 2x'^2) \\ \Delta t_y &= 2k_4 x'y' + k_3(r^2 + 2y'^2) \qquad (6.3)\end{aligned}$$

  k_3, k_4 are known as the *tangential distortion coefficients.* The tangential distortion is due to "decentering", or imperfect centering of the lens components and other manufacturing defects in a compound lens.

- **Affine distortion**

$$\begin{aligned} undist_x &= f_x(x' + \Delta r_x + \Delta t_x + \alpha(y' + \Delta r_y + \Delta t_y)) + c_x \\ undist_y &= f_y(y' + \Delta r_y + \Delta t_y) + c_y \end{aligned} \tag{6.4}$$

  α is known as the *skew coefficient*, and can account for non-rectangular pixel sensors.

$(undist_x, undist_y)$ is finally the undistorted image coordinate of the original point (x, y).

All these parameters can be estimated semi-automatically by using the MATLAB toolkit provided by the website authors. Normally, 10 to 15 shots of a reference checkerboard pattern are enough to give a good estimate for most of the parameters. Figure 6.1(b) on the preceding page shows the result of a run for one of our wide-angle cameras. The arrows in the top of the figure show the displacement of the pixels. A pixel that should lie at the start of an arrow under optimal conditions is distorted to the end of it by the lens. The small circle in the center depicts the estimated center of projection vs. the ideal center depicted as a small cross. All estimated parameters (complete with error bounds) can be seen in the bottom half of the figure.

6.1.1 Reduced distortion models

As stated on Bouguet's website, a reduced model is often sufficient for modern CMOS and CCD cameras According to it, modern lenses do not exhibit noticeable tangential distortion, and are usually using a rectangular pixel sensor which makes skew correction unnecessary. (In fact, a second radial order only model can often give a fair approximation for small cameras).

After doing some tests with reduced distortions models, it became clear that there are almost no performance gains when using a reduced model on modern computers – and as we'll see later on, the model only needs to be evaluated once in a preprocessing step. So we opted for implementing the full distortion model supported by the MATLAB toolkit anyway.

6.2 Perspective correction

Ideally, the camera would be mounted directly above the middle of the field, and in a height that uses the whole image area to capture the playing field. Unfortunately, this often can't be done in practice. It's complicated enough to exactly align the camera in the lab, but under competition conditions it's next to impossible. Competitions are usually quite busy, the field will be moved or may still be under construction, and often there is no possibility to mount the camera above the field. As a compromise, a camera stand at the side of the field is often the only viable option for getting pictures from the field.

Taking pictures from non-optimal positions will always introduce perspective warping and foreshortening. While perspective texture mapping is widely used in computer graphics, we need to invert the process, and basically undo the mapping that is introduced by the camera's position.

Heckbert [11] gives a nice introduction into the different types of texture mappings. We will use a 2-D projective mapping to model the perspective distortions of our ceiling camera.

Projective mappings can be written as rational linear mappings:

$$x = \frac{au + bv + c}{gu + hv + i}, \quad y = \frac{du + ev + f}{gu + hv + i} \tag{6.5}$$

where (u, v) is a source pixel in camera image coordinates, and (x, y) receives the undistorted pixel coordinates.

Or more easily in 2 dimensional homogenous matrix notation:

$$(x', y', w) = (u', v', q) \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \tag{6.6}$$

where $(x, y) = (x'/w, y'/w)$ for $w \neq 0$, and $(u, v) = (u'/q, v'/q)$ for $q \neq 0$. (See [10] for a complete discussion of homogenous coordinates).

As can be seen in Equation 6.6, a perspective texture mapping has 9 degrees of freedom $(a \ldots i)$. Because any linear multiple $\neq 0$ of a homogenous point maps to the same point in normal coordinates, we can reduce the equation to 8 degrees of freedom by assuming $i = 1$ without loss of generality.

In our application, we choose to implement an interactive texture mapper. The user can define a source and destination quadrilateral, to align the image of the playing field to the desired coordinates. The two quadrilaterals consist of four coordinates each, which yields 8 user-set parameters in total.

To infer the 8 parameters of a projective mapping from these coordinates, a 8x8 system of equations can be set up and solved by gaussian elemination or similar methods. However, because we wanted to avoid the subtle complexity of implementing an equation solver, we opted for an easier composition based method as introduced in Heckbert's paper.

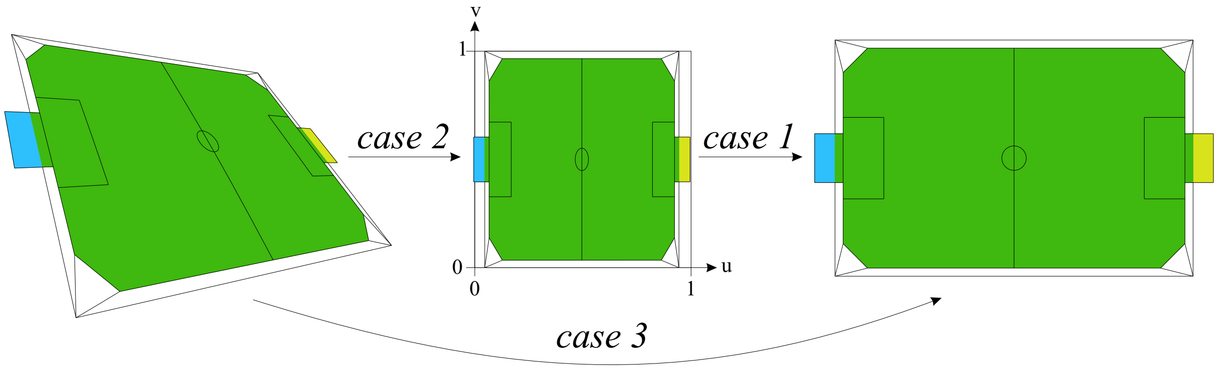


Figure 6.2: Texture mapping as a composition of simpler mappings

Projective mappings define a bijective mapping between two coordinate spaces. Even more remarkable is the fact that the inverse of a projective mapping, is a projective mapping. The inverse transform can simply be calculated by taking the inverse of the homogenous matrix.

Because of this, every projective mapping between two arbitrary quadrilaterals can be decomposed into a mapping from the source quadrilateral to a unit square, and then from a unit square to the destination quadrilateral (see Figure 6.2).

Fortunately the parameters for the first case (unit square to quadrilateral) can be directly calculated by algebraic methods without any explicit equation system solving. The second case (quadrilateral to unit square) can be solved in exactly the same way, but with an additional matrix inversion at the end of the calculation. The final composite third case (quadrilateral to quadrilateral) is then calculated by a matrix multiplication of the first two cases.

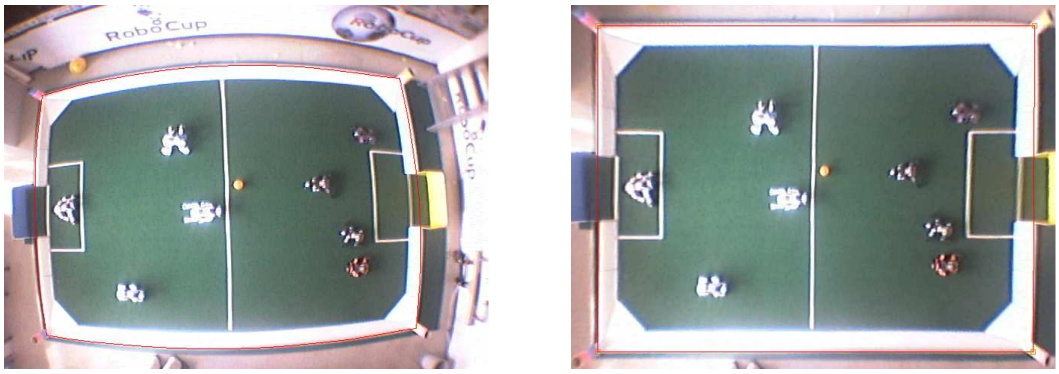

(a) Original image (b) Corrected image

Figure 6.3: Ceiling camera image of the playing field

6.3 Implementation

After all the equations and complexity of lens distortion and perspective correction, the most important question was, if all this can be implemented efficiently enough for realtime processing. Calculating a complete lens- and perspective corrected image takes approximately 100ms for a 640x480 image on a 3 GHz Pentium IV. While this seems to be quite fast, it isn't fast enough to keep up with the framerate of modern cameras.

Luckily, the displacement map doesn't change after the intrinsic and extrinsic parameters are set and can thus be calculated in a preprocessing step. The ceiling camera view can then use this map to do the actual warp (which is much faster than re-evaluating the formulas for each pixel).

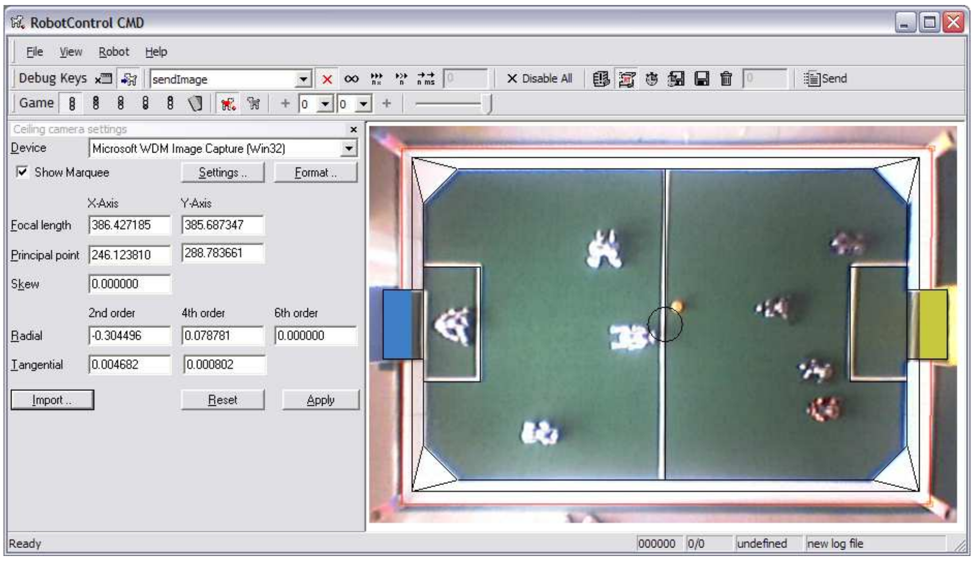
6.4 User interface

One of the main goals was an easy to use interface. Working with the ceiling camera should be easy and fun, because it will be one of the building blocks for the work of the following project group. As mentioned in the previous sections, we chose an interactive warping approach[1].

The ceiling camera view is fully integrated into the RobotControl application. All relevant settings can be done in the *Ceiling camera settings* dialog, which can be found in the *View* menu (see Figure 6.4 on the next page).

The context menu of the playing field view has been extended with a ceiling camera entry. Enabling this item will show the undistorted camera image superimposed on the playing field.

[1] Heckbert already suggested such an interactive application in his original paper, but the computing power was too limited to actually implement this on consumer hardware in 1989.



Ceiling camera settings

Device	Microsoft WDM Image Capture (Win32)		
☑ Show Marquee		Settings ..	Format ..
	X-Axis	Y-Axis	
Focal length	386.427185	385.687347	
Principal point	246.123810	288.783661	
Skew	0.000000		
	2nd order	4th order	6th order
Radial	-0.304496	0.078781	0.000000
Tangential	0.004682	0.000802	
Import ..		Reset	Apply

Figure 6.4: Integration of the ceiling camera user interface in RobotControl

The left side of the screenshot shows the actual camera settings dialog. The user can select a camera connected to the computer, and can adjust its settings (like brightness or frame rate) and image format by using the corresponding buttons. After a camera has been selected, the user can enter the intrinsic distortion parameters by using the edit boxes or, more conveniently, by importing a parameter file generated by the MATLAB toolkit.

After the intrinsic parameters have been set, a click on *Show marquee* will enable the perspective warp edit marquee (the red quadrilateral with the four orange handles as shown in Figure 6.4). The parameters of the perspective warp are completely controlled by interacting with the marquee. The four orange handles are called *reference points.*

Typically the user will first move the reference points to some easily identifiable features in the camera image. This can be done by dragging the reference points while holding down one of the `shift` keys on the keyboard. The playing field corners, or the goal posts are some good points to select.
In the second step, the user will drag the reference points to the corresponding points of the debug drawing. This can be done by simply dragging around the reference points without holding down any modifier keys.

These two steps can be iterated until the camera image and the debug drawing match up against each other as shown in Figure 6.4.

Unfortunately it's possible to get stuck in a dead end, when some of the reference points are moved very close to each other, or form a degenerate quadrilateral (like a point, line or triangle). The reset button in the ceiling camera settings dialog will help in these cases, by resetting the marquee to a default mapping.

6.5 Towards an automated oracle

As laid out in the first section of this chapter, the ceiling camera overlay is only the first step in leveraging the ceiling camera for the RoboCup. The second step is the development of an automated oracle, that can detect the robots' position and orientation in the camera image, and provide this information to the actual robots on the field. Because writing the camera overlay did take far more time than expected, this work is left for the following project group.

We think that it would be best to develop a stand-alone application that can run on one of the servers of our arena, and provide the oracle data over a client/server protocol. Some parts of this task are already finished – an early stand-alone prototype of the ceiling camera overlay was developed by using the Intel IPL[14], and some functions taken from the OpenCV library [15]. The next project group should be able to back-port some of the functions from the integrated RobotControl version to this stand-alone program without much effort (the intrinsics dialog, MATLAB import functionality and perspective marquee should be easily reusable).

When we worked with the ceiling camera overlay, it became clear, that the quality of normal consumer grade webcams like our Philips QuickCam Pro (already one of the best models on the market) might not be sufficient for automated image processing. The robots often appear as a greyish blob, without any easily detectable features. The poor quality of cheap wide-angle lenses combined with the small image sensor size of the camera and video compression needed for slow USB 1.1 connections contribute to the bad quality.

The next project group should evaluate industrial quality image processing cameras, that are readily available with $^1/_2$" image sensors, USB 2.0, and resolutions in excess of 1280x1024 pixels at 50 frames per second. Some care must be taken when choosing a wide-angle lens, because at the time of this report some rule changes concerning enlarging the field where in active discussion.

Chapter 7

Competitions

7.1 German Open 2004

The German Open[1] is an international competition of Robot Soccer. This year it has taken place in the Heinz-Nixdorf-Museum in Paderborn from 01.April 2004 - 04.April 2004. Many teams took part in the following leagues:

- Middle Size League
- Small Size League
- SONY-Legged League
- Rescue Simulation League
- Soccer Simulation League
- Junior League

In addition to the Microsoft Hellhounds, 7 other teams competed in the Sony-Legged League. Some of them with the new ERS-7, some teams with the old ERS-210 and some teams even had both kind of robots in their team. It can be suggested that the new robots are head and shoulders above the old ones, which is true both physically and metaphorically, but in table 7.1 on the following page it can be shown that local solutions can be more important than expected at first sight: The Hamburg Dog Bots used the old ERS-210 only, but their performance was better than expected, since they played with the GT2003-Code from the last year, which was only slightly modified.

The *dynamic team tactics* (see 5.1 on page 56) were used for the first time at a competition. One of the most apparent and noticeable features of the *dynamic team tactics* at the German Open 2004 was the behaviour around the ball.
The situation of two or more robots from the same team fighting for the ball would seldomly occur, instead of that teammates walked to opposite areas of the field, intending to receive a pass. During the games though it turned out that the *dynamic team tactics* have not been tuned enough. The main problem was the decision time the robots needed to determine which robot was to move to the ball. Although the robots saw the ball

[1]German Open official Site: http://www.ais.fraunhofer.de/GO/2004/

and calculated so called ball percepts, which was indicated by their shaking tails, several seconds were wasted sometimes.

However, in the preliminary round the Microsoft Hellhounds had to play against the later champion and the later third placed team, nevertheless they could achieve a great 4th place overall. A close defeat in the semi-finals (3:4 against the later 2nd placed Darmstadt Dribbling Dackels[2]) stopped the run to the final.

Competition-results:

Games	Team 1	Score	Team 2
Round Robin	**Microsoft Hellhounds**	4 : 0	Les Trois Mousquetaieres
Round Robin	Microsoft Hellhounds	0 : 6	**Hamburg Dog Bots**
Round Robin	Microsoft Hellhounds	1 : 2	**Aibo Team Humboldt**
Quarter Finals	**Microsoft Hellhounds**	4 : 3	Bremen Byters
Semi Finals	Microsoft Hellhounds	3 : 4	**Darmstadt Dribbling Dackels**
Game for 3rd Place	Microsoft Hellhounds	0 : 6	**Hamburg Dog Bots**

Table 7.1: The results of the Team Microsoft Hellhounds at the German Open 2004

7.2 Opens 2004

Beside the German Open in early April, the Microsoft Hellhounds participated in three further national opens. The Australian Open on April 16th, the US Open from April 24th to 27th and the Japan Open from May 1st to 4th. As representative and field operator we sent André Osterhues to the Australian and to the Japan Open. Walter Nistico, Arthur Cesarz and Bernd Schmidt went to the American Open.
During these events a team of several people stayed awake in the nights and supported the people at the chmpionships via the internet. We earned much experience from this and we were able to improve our code as a result of the competition with the international teams. Without this experience we would have had much more problems during the world championship RoboCup 2004.
At the Australian Open we achieved the third place in the competition, at the American Open we achieved the fourth place and at the Japan Open we achieved a second place in the technical challenge.

7.3 RoboCup 2004

By the year 2050,
develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.

This is the topmost milestone of the international research and education initiative RoboCup. Its main objective is to provide a standard problem where a wide range of technologies can

[2]member of the GT2004 Team

be examined and integrated. Playing soccer is a new benchmark for artificial intelligence and robotic engineering and replaces the chess benchmark of the last decades.
Furthermore the annual world championships provide an opportunity to meet with scientists and robotic engineers from around the world to compare and share the newest developments in robotic science. In addition to this RoboCup tries to inspire and entice children on the topic of robotics and computer science with *RoboCup Junior*.
This year the championships took place in Lisbon, Portugal. With 346 teams from 37 countries and a total of 1600 participants a new record number in the history of RoboCup was achieved. We participated in the Sony 4-legged Robot League as part of the GermanTeam which is constituted by members of the universities of Berlin, Bremen, Darmstadt and Dortmund.

7.3.1 WE ARE THE CHAMPIONS!!

Twenty-four teams were registered for the RoboCup 2004 Tournament of the Four-Legged-League. The organisation divided them into four groups with six teams each. Group A consisted of only five teams because team "Wright Eagles" did not take part and all matches against this team had been canceled. Five games had to be played in the round robin for each team. After these games the two best teams of each group had to participate in the play-offs (see 7.2).

Group A	Group B	Group C	Group D
ARAIBO	Hamburg DogBots	CMPack'04	ASURA
Les 3 Mousquetaires	Jollie Pochie	Dutch Aibo Team	Baby Tigers
UChile 1	Metrobots	FC Portus	Georgia Tech
Upennalizers	Nubots	Mi-Pal	GermanTeam
UT Austin Villa	SPQR	TecRAMS-Mexico	rUNSWift
-	UW Huskies	UTS Unleashed!	Team Chaos

Table 7.2: The drawing of Round Robin of the Four-Legged-League of RoboCup 2004 in Lisbon

Round Robin: Game One

For the GermanTeam the tournament started with a major game against the former world champion rUNSWift. After the first half, which was overshadowed by technical problems on both sides, the Australian team led with 2:1 goals. During halftime the problems were solved and the GermanTeam won with 4:2 goals after a tense second half. This victory was important for the whole team. One of the major opponents had been defeated and one big step had been done towards the quarter final.

The game showed several problems of the team play and the team behavior. After the game a team meeting was held to analyze and solve the problems. The team members were splitted into groups to work on solutions. The GermanTeam focused on the image processing, the kickengine, the team behavior and the goalkeeper behavior.

- **Vision and Imageprocessing:** The imageprocessing indicated problems in the ball recognition. The ball was good recognized in the distance, but close to the robot it showed problems when the ball disappeared from sight. The main problem was estimating the direction the ball was rolling to, which was needed to calculate its estimated position. The localisation also showed weaknesses, but with an improved color table these problems were solved eventually.

- **Kickengine:** The game against rUNSWift revealed several situations where the GermanTeam robots could make a good kick. However, the kick engine did not activate a kick, although an appropriate kick was registered for most of these occasions. Again training sessions had to be made to solve this problem. Thereby it was shown how important a realistic environment was for these tests. For example it was important to use the maximum walking speed for approaching the ball before the kick. Special kicks for specific situations in the game were missing. Additional kicks had to be created for these situations and trained and registered for the kickengine.
- **Team behavior:** The positioning of the robots left space for improvements. In some situations of the game the robots did not get a chance to score because of wrong positioning, especially when the opponent was outnumbered in his own half. The transitions between the defensive and the offensive behavior were not efficient enough. And in some situations the worse positioned robot tried to get to the ball instead of the better positioned Aibo. This mainly occured, when robot A was closer to the ball than robot B, but was obstructed by an opponent robot. The optimal choice would have been, that robot B tried to get to the ball. Instead, robot A kept wrestling with the opponent's robot, while B idly waited in a pass receiving position.
- **Goalkeeper behavior:** The goalkeeper had difficulties with its localization. This problem occurred because initially the GT2003 Self Locator was used. In the first game the goalkeeper several times stood too far behind the goalposts to see the landmarks correct. With the completion and introduction of the GT2004 Self Locator, this problem vanished. Another task was to improve on the reflex behavior. The reflexes help the goalkeeper to react to fast incoming balls. The robot had to quickly spread its legs for a short time to intercept balls, which would else roll into the goal. The reflex behavior worked in the game but showed space for improvements concerning optimal timing. The third task of the goalkeeper behavior was its general behavior. The robot showed weaknesses in clearing the ball in front of the goal. Optimal timing was the main issue here as well.

Round Robin: Game Two

In the second game the GermanTeam met Team Chaos and defeated them with 13:0 goals. In this game the differences between the ERS-210 and the ERS-7 were clearly shown. Because of the stronger motors of the ERS-7 against the ERS-210, it is faster and kicks stronger. In the game this was proven by the GermanTeam robots. They reached the ball faster every time, even when the opponent had kick-off. The team behavior showed improvement since the first game. The Robots positioned themselves better and took advantage of the weaknesses of the opponent team.

Round Robin: Game Three

The third game was against the winner of the Japan Open 2004: Asura. The game was expected to be as challenging as the first game against rUNSWift. But the opponent robots did not come into play against the GermanTeam behavior. The game ended 6:1 for the German Team.

Round Robin: Game Four

Georgia Tech Yellow Jackets was the opponent of the fourth game. The American team showed an overall poor performance. At halftime the score was 6:0 for the GermanTeam. Because of technical problems and their inferiority against the GermanTeam they decided to concede and gave up in second half. The tournament officials ruled to count the second half as the first one, according to the official tournament rules. So, the final score of this game was 12:0 for the GermanTeam.

Round Robin: Game Five

The last game of the group was against the Japanese team Baby Tigers. The defensive skills of the Baby Tigers were a challenge for the GermanTeam robots. But with the kickengine improvements and the team behavior the game ended with 7:0 goals for the GermanTeam.

Group A	Group B	Group C	Group D
1. Upennalizers	**1. Nubots**	**1. UTS Unleashed!**	**1. GermanTeam**
2. UT Austin Villa	**2. Hamburg DogBots**	**2. CMPack'04**	**2. rUNSWift**
3. ARAIBO	3. Jollie Pochie	3. FC Portus	3. ASURA
4. Les 3 Mousquetaires	4. UW Huskies	4. Dutch Aibo Team	4. Baby Tigers
5. UChile 1	5. SPQR	5. Mi-Pal	5. Georgia Tech
-	6. Metrobots	6. TecRAMS-Mexico	6. Team Chaos

Table 7.3: The results after Round Robin of the Four-Legged-League of RoboCup 2004 in Lisbon. Highlighted teams were qualified for the Quarter-Final.

Quarter-Final

As the winner of Group D the GermanTeam had to play against the second of Group C, CM Pack'04 (see 7.3). This game was considered as the revenge for the quarter final of the RoboCup 2003 in Padova. In that match, the Americans stopped the GermanTeam after a tense thirty minute long penalty shootout. This game was different, though. The GermanTeam was superior and got in the lead by two goals early in the game. CMPack0́4 used a timeout for changing the game sticks for a different set. This did not show the expected effect and the game ended 9:0 for the GermanTeam. The American team did not get in front of the GermanTeam goal long enough to kick, but the offensive behavior of the GermanTeam was by far superior against the American goalkeeper. The GermanTeam attended the semi-final of the RoboCup competition for the first time in their third RoboCup participation. The main goal of the GermanTeam was achieved (see 7.4 on the following page).

Upennalizers	4:1	Hamburg DogBots
UTS Unleashed!	9:1	rUNSWift
Nubots	6:5	UT Austin Villa
GermanTeam	9:0	CM Pack'04

Table 7.4: The results of the Quarter-Final of the Four-Legged-League of RoboCup 2004 in Lisbon. Highlighted teams were qualified for the Semi-Final.

Semi-Final

The semi-final against the Australian team Nubots began tight as expected. The Nubots got early in the lead with the 1:0. But after this shock the GermanTeam behavior showed its strength. The offensivly playing robots turned the game. The new goalkeeper behavior showed its skill. Many times the goalkeeper cleared the ball from the GermanTeam goal. The timing of reflexes and ball clearance was nearly optimal and the localisation of the goalkeeper in front of the goal was perfect all the time. Because of these improvements the Australian team kicked the ball into the German goal only once. The semi-final ended 9:2 for the GermanTeam and earned respect by the opponent UTS Unleashed! of the upcoming final (see 7.5).

Upennalizers	1:5	**UTS Unleashed!**
Nubots	2:9	**GermanTeam**

Table 7.5: The results of the Semi-Final of the Four-Legged-League of RoboCup 2004 in Lisbon. Highlighted teams were qualified for the Final.

Final

As all participants of the RoboCup Four-legged league agreed, the final was held between the two best teams of the tournament. The Australian Team UTS Unleashed! was the winner of the Australian Open 2004 and like the GermanTeam winner of their group in Round Robin. Both teams were undefeated in this tournament, both teams led the game statistics, most goals scored and the goals scored/conceded ratio. Like in the semi-final the Australian team got in the lead with 1:0. Just as in the semi-final the GermanTeam could equalize the score. This game developed into the most exciting game of the championship, as UTS Unleashed! got in the lead again with 2:1. Just before halftime the Germans scored and the game was even again. The tight score provided tension for the second half. In this half the Germans got a better start than the Australians. For the first time of this game the GermanTeam got in the lead with 3:2. The robots of UTS Unleashed! failed several times against the German goalkeeper and the German robots scored again. After this 4:2 the Germans got several chances for the 5:, but they failed. The Australian defence proved their skill by standing well positioned. With a felicitous pass-play, which they already showed in their demonstration for the OpenChallenge, they started the kick to the 4:3. The German lead was decreased to one goal, but the robots of the GermanTeam quickly retaliated and scored again for the final 5:3. The last minutes showed no advantage for any team. After the game ended the GermanTeam became World Champion 2004 of the RoboCup (see 7.6 on the following page, 7.7 on the next page and 7.1 on page 84).

Upennalizers	4:5	**Nubots**

Table 7.6: The result of the match for third place of the Four-Legged-League of RoboCup 2004 in Lisbon.

UTS Unleashed!	3:5	**GermanTeam**

Table 7.7: The result of the Final of the Four-Legged-League of RoboCup 2004 in Lisbon. The GermanTeam won the World Championship.

7.3.2 Open Challenge

Next to the soccer competitions in the Sony 4-legged Robot League the teams competed also in some technical challenges. One of these challenges was the newly introduced *open challenge* which was intended to enable the teams to present and demonstrate parts of their research in a creative and entertaining way.
The university of Dortmund was responsible for the contribution to the open challenge for the GermanTeam. We decided to create a scenario which should demonstrate our research on cooperative behavior and came up with the following idea:
One of our robots should kick a goal with a ball from the Mid-Size League of RoboCup. To accomplish this task, the robot has to cooperate with four more robots of our team and has to *transform* thereby into a virtual Mid-Size League robot.
For this transformation we built a cart with rails at the sides and a ramp to climb on the cart. The main robot stays on the top of the cart, meanwhile four other robots were staying at the sides of the cart biting into the rails. In this manner we literally visualized the virtual robot metaphor used by us so far.
The robot staying on the top of the cart localized itself, searched for the orange ball and generated the walking requests for the four cart-moving robots. These robots were not able to localize for themselves since they were looking directly towards the sides of the cart. In this situation they were simply representing actuators of the main robot.
Almost every team in Lisbon which saw this demonstration was truly impressed and entertained. Accordingly we achieved the first place in the open challenge competition by a great margin.

Figure 7.1: GermanTeam members in Lisbon

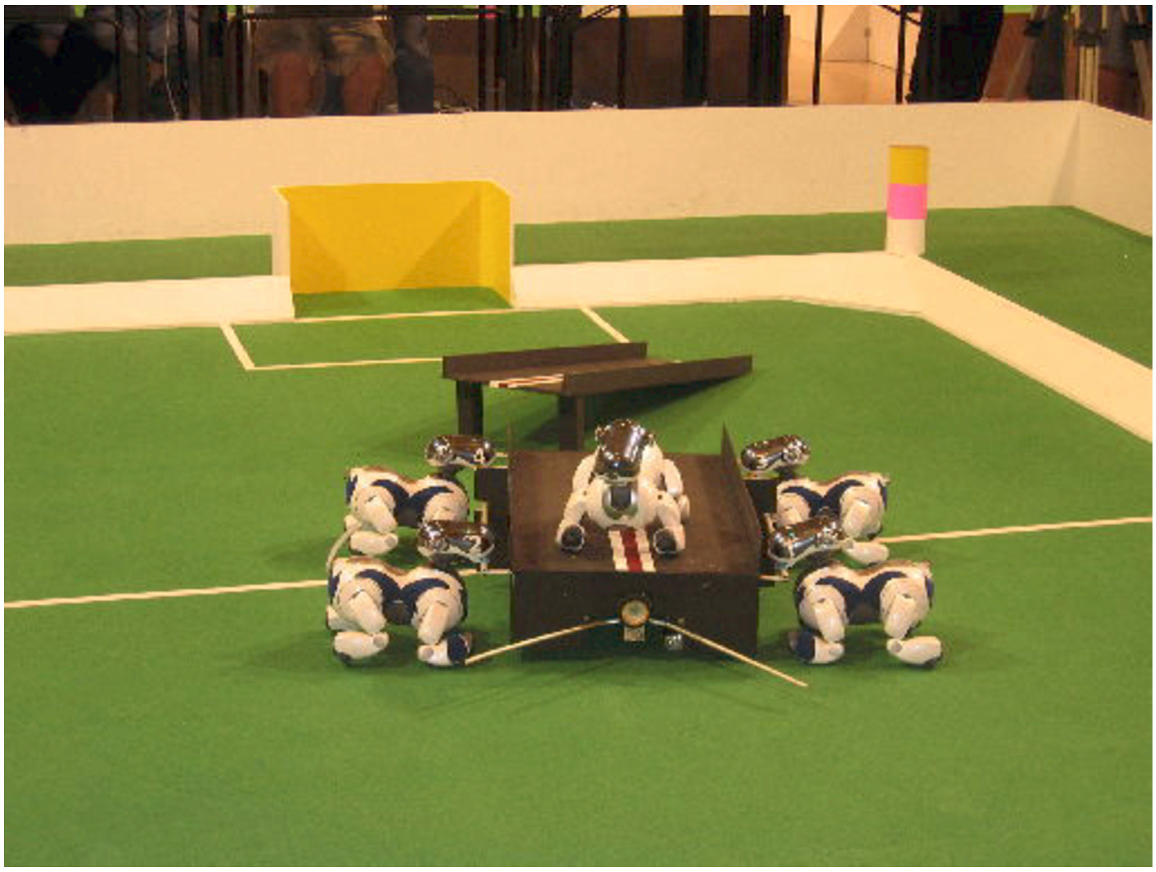


Figure 7.2: The Open Challenge contribution of the german team

Chapter 8

Side Projects

8.1 World State Player

It turned out in the project group, that the development of an efficient behavior is an important and substantial part of the work. Next to evolving new ideas for the behavior the implementation requires extensive tests and error analyses. Specially the analysis of the results of the tests is difficult to manage, because of the many sources of errors which the behavior offers. In addition to errors in strategy or general errors in the code the behavior cause problems with wrong sensor interpretation. For the development of an efficient behavior it is important to verify the correct evaluation of sensor data the Aibo gets during tests. A wrong analysis can pretend or cover bugs in the strategy. According to this problem it would help the developer to see all important sensor and world data together on the screen. The comparison of these data enables the estimation of possible causes of errors in the behavior. A correct interpretation of the sensor and world data eliminates one of the possible sources of errors.

Robot Control provides the Log-Player. The Log-Player supports recording of logfiles on the Aibo and playing them in Robot Control. Every sensor and world data of the robot can be stored in a logfile and for this reason be used for debugging. This debug-data is called on Robot Control with debug keys. The Log-Player is limited to record and play the logfile of the current selected robot instead of all playing robots. This is satisfactory for example to calibrate a new colortable, but according to the description above it is inadequate for debugging a new behavior.

For behavior debugging an enhanced logfile player is needed which maintains the support of up to four logfiles. This extension would cover a whole team which is needed for developing the teambehavior. The addition of support for the four opponent Aibos is not needed because the development of a new behavior is tested in games against the old tested behavior of last year. Furthermore, comparison with reality is helpful for the developer. It helps finding errors in the world model. In some occasions all Aibos suffer from the same error and without an image of the real world model these errors would be undiscoverable. According to this it is necessary to have a video of the testgame recorded. Helpful is a ceiling camera which provides an overview of the game. Additionally the world model data in Robot Control is shown from a similar point of view. The comparison with a ceiling camera would be much easier than with a normal camera which only films parts of the playing field.

The project group attended to this problem and developed the World State Player. The

World State Player is an enhancement of the Log-Player. The World State Player supports the recording and playing of up to four different logfiles. In addition the play back of a video and a merged world state of all four logfiles is maintained. The merged world state is helpful for the comparison with the video. It provides the same view as a ceiling camera and shows all world model data on the field, but many times this field is too complex. According to this each logfile is also shown separately on a separate field in the window. The World State Player provides a toolbar with additional functions. Next to playing and recording it supports the skipping of frames in the logfiles and video and a slidebar for direct point access in the logfile. Next to the toolbar four textfields contain the current time stamp of each logfile.

8.2 Demo Stick

Nowadays science consists not only of research and publications, but also of explaining and legitimating itself towards the public. Additionally and in correlation to those public relations young people have to be enthused for science to build the next generation of scientists. Therefore we have to present our research topics in a more attractive and pictorial way, abstracting from scientific complexity and details and using more or less self explaining demonstrations. These were the reasons that led us to the creation of the *demo stick*[1]. It is a non interactive program which shows some ball handling abilities of our robots. The space needed by the robot when executing the demo stick is rather small, thus it can be run on educational events, press conferences, exhibitions, etc.
As an eye-catcher the robot is able to put the ball on his back and to juggle with the ball in this position. Since all the spectacular movements are localized it makes it easy for photographers to get some good shots for their newspapers or tv-reports.
Of course a predefined sequence of movements, even if it is quite spectacular, isn´t sophisticated science. But pictures which illustrate headlines of our field of research, like *teaching robots to play soccer*, can help to get the attention needed to gather interest in the public.

8.3 Walking on a Leash

Another nice demonstration of our research is a robot *walking on a leash.* Actually the used technique is a spin-off of the open challenge. In the challenge we used a kind of feedback loop to stabilize the head position of a robot while biting into the rail of the cart and moving. Used for *walking on a leash* the dog starts turning when the leash and thus the head is moved to one side. Additionally we extended this feedback loop by a vertical component which is used to control the walking speed of the robot. As the head is lifted by pulling the leash the robot increases its speed and if the leash is released and thus the head is lowered the robot decreases its speed again.
Like the demo stick *walking on a leash* is a beneficial demonstration to draw someone's attention to the field of robotic research and especially to encourage children to focus on robotics and science in general.

[1]a *stick* is the memory media of our robots

References

[1] Jean-Yves Bouguet. *Camera Calibration Toolbox for Matlab*, 2004.
`http://www.vision.caltech.edu/bouguetj/calib_doc/`.

[2] J.E. Bresenham. *Algorithm for Computer Control of Digital Plotter. IBM Systems Journal, Vol. 4, No. 1*, April 1965.

[3] D.C. Brown. *Lens Distortion for Close-Range Photogrammetry. Photometric Engineering, pages 855-866, Vol. 37, No. 8*, 1971.

[4] J. Bruce, Tucker Balch, and Maria Manuela Veloso. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '00)*, volume 3, pages 2061 – 2066, October 2000.

[5] H.D. Burkhard, R. Brunn, I. Dahm, U. Düffert, K. Engel, D. Göhring, J. Hoffmann, M. Jüngel, M. Kallnik, M. Kunz, M. Lötzsch, A. Osterhues, S. Petters, M. Risler, C. Schumann, M. Stelzer, O. von Stryk, T. Röfer, M. Wachter, and J. Ziegler. *GermanTeam 2003 - Team report.*
`http://www.robocup.de/germanteam/GT2003.pdf`.

[6] J. Canny. *A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6*, November 1986.

[7] T.A. Clarke and J.G. Fryer. *The Development of Camera Calibration Methods and Models. Photogrammetric Record, 16(91): pages 51-66*, April 1998.

[8] Sebastian Deutsch, Thomas Dickhöfer, Wenchuan Ding, Kai Engel, Piotr Kudlacik, Andre Osterhues, Jan Prünte, Andreas Reiß, Sebastian Schmidt, Christian Thiel, and Michael Wachter. *Sony Legged League: Entwicklung von verteilten Algorithmen zur effizienten Kontrolle von autonomen Fußballrobotern.*
`http://www.m-wachter.de/endbericht.pdf`.

[9] Robert B. Fisher. *CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision*, 2004.

[10] James D. Foley and Andries van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

[11] Paul S. Heckbert. *Fundamentals of Texture Mapping and Image Warping.* Master's thesis, University of California, Berkeley, CA94720, June 1989.
`http://www-2.cs.cmu.edu/~ph/texfund/texfund.pdf`.

[12] Martin Lötzsch. *xabsl - The Extensible Agent Behavior Specification Language.*
`http://www.ki.informatik.hu-berlin.de/XABSL`.

[13] Ingo Rechenberg. *Evolutionsstrategie '94*. Frommann–Holzboog, Stuttgart, 1994.

[14] Intel Research. *IPL – Image Processing Library*, 2004.
`http://developer.intel.com/software/products/ipp/`.

[15] Intel Research. *OpenCV – Open Source Computer Vision Library*, 2004.
`http://www.intel.com/research/mrl/research/opencv/`.

[16] Ingo Wegener. *Grundvorlesung Datenstrukturen*, 1992.