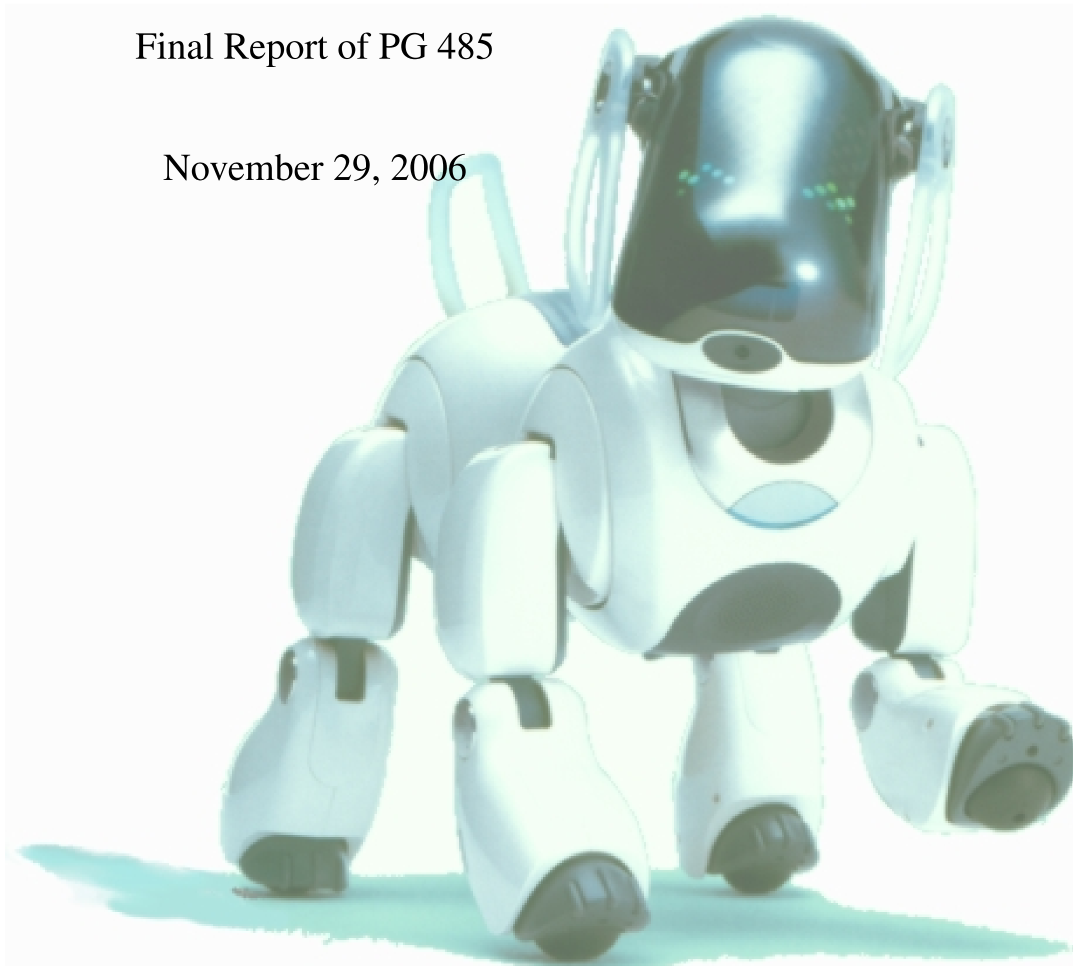
UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Thorsten Beuchel David Fiedler
Jens Kaufmann Michael Kruse Firas Nasched
Tina Neng Jan Rottberg Moritz Schallaböck
Benjamin Schwertfeger Adam Vincze
Jens Wege Marcel Witte

Robot Soccer
4-Legged-League

Final Report of PG 485

November 29, 2006



GERMANY · D-44221 DORTMUND

Contents

1 Introduction **11**
1.1 Playing Soccer with 4-Legged-Robots . . . 11
1.1.1 Intentions . . . 11
1.1.2 Hardware . . . 11
1.2 Innovations in 2006 . . . 13

2 ImageProcessor **15**
2.1 Ball Detection Improvements . . . 15
2.1.1 Problem Description and Motivation . . . 15
2.1.2 Solution Approach . . . 15
2.1.3 Description of the chosen solution . . . 17
2.1.4 Discussion of the solution's quality . . . 18
2.2 Motion Compensation Calibration . . . 19
2.2.1 Description of the current system . . . 19
2.2.2 Motivation for the Calibration . . . 19
2.2.3 Error and Calibration of the Motion Compensation . . . 19
2.2.4 Results of the Calibration . . . 20

3 Horizon Calibration **23**
3.1 Problem Depiction . . . 23
3.2 Different Approaches . . . 23
3.2.1 DLT-Method . . . 25
3.2.2 Automatic Horizon Detection . . . 26
3.2.3 Manual Calibration . . . 27
3.2.3.1 Horizon-Based Method . . . 27
3.2.3.2 Reference-Point-Based Method . . . 27
3.3 Evaluation and Achieved Result . . . 28

4 Ball Locator **31**
4.1 Goals of Ball Modeling . . . 31
4.2 Possible Solution Attempts . . . 31
4.2.1 Particle Filter . . . 31
4.2.2 Kalman Filter . . . 32

4.3 Chosen Solution: *MSH2006BallLocatorKalman* 32
4.3.1 Basic Concept 32
4.3.2 Covariance Matrices by Measurements 33
4.3.2.1 Position Variances 34
4.3.2.2 Velocity Variances 36
4.3.2.3 Conclusion 37
4.3.3 Additional Improvements 38
4.3.3.1 Distance Offset 38
4.3.3.2 Non-Linearities 39
4.3.3.3 Velocity Filtering 39
4.3.3.4 Experimental Forecast Method 40
4.4 Evaluation of Results 40

5 Opponent player modelling **43**
5.1 Goal and Challenges 43
5.2 Particle Filters 44
5.3 Clustering 46
5.4 Single Filter Modelling 46
5.5 Multiple Filter Modelling 48
5.6 Results 50

6 Behavior Learning **53**
6.1 Specification of the Problem 53
6.1.1 Motivation for Learning on a Robot 53
6.1.2 Optimization of the Behavior “grab-ball” 53
6.2 Different Learning Methods 54
6.2.1 Reinforcement Learning 54
6.2.2 Evolutionary Strategies 54
6.2.3 Experimental Search Space Exploration (ESSE) 55
6.3 Description of the Chosen Approach 56
6.3.1 Behavior Evolution with Reinforcement Learning 56
6.3.2 Parameterization of the Behavior “grab-ball” 56
6.3.2.1 Calculation of the Fitness Function 57
6.3.2.2 Training Procedure 58
6.3.3 Optimization strategies 58
6.3.3.1 (1+1)-Evolution Algorithm with 1/5th-Rule for Mutation Strength 58
6.3.3.2 Experimental Search Space Exploration (ESSE) 59
6.3.4 Button Interface for Manual Award and Mutation Direction 59
6.3.5 Other Improvements 60
6.4 Evaluation of the Results 60
6.4.1 Concrete Analysis 60
6.4.2 Comparison of the Implemented Optimization Strategies 61

6.4.3 Conclusion 61

7 WalkingEngine 63
7.1 Description of the existing WalkingEngine 63
7.2 Problems of the existing solution 64
7.3 Approaches for the correction of the WalkingEngine 66
7.4 Correction by measurement 67
7.4.1 The WalkCalibration tool 68
7.5 Correction by odometry table inversion 69
7.6 Correction by optimization of polygonsets 69
7.6.1 Manual Optimization 70
7.6.2 WalkshapeViewer 70
7.6.3 Evolutionary Optimization 72
7.6.4 Customize Odometry 72
7.7 Conclusion 73

8 Special Actions 75
8.1 Significance of Special Actions 75
8.2 Creating, Editing and Using Special Actions 75
8.3 Experiences 76

9 Tools 77
9.1 Robot ControlXP 77
9.1.1 Motivation 77
9.1.2 Fundamental structure 78
9.1.3 Framework 78
9.1.4 Manager 78
9.1.5 UserControls 78
9.1.6 Messages 79
9.1.6.1 Debugging on the robot 79
9.1.6.2 Extended Debug Mechanism 80
9.1.6.3 DebugData 81
9.1.6.4 Messages 81
9.1.6.5 Poll (RobotControl ⇒ robot) 81
9.1.6.6 idDebugResponse (robot ⇒ RobotControl) 81
9.1.6.7 idDebugRequest (RobotControl ⇒ robot) 82
9.1.6.8 idStreamSpecification (robot ⇒ RobotControl) 82
9.1.6.9 idDebugDataResponse (robot ⇒ RobotControl) 82
9.1.6.10 idDebugDataChangeRequest (RobotControl ⇒ robot) 83
9.1.6.11 idModuleSoulutionTable robot ⇒ RobotControl) 83
9.1.7 Framework 83
9.1.7.1 ConnectionUserControls 84
9.1.7.2 ControllerUserControl 84

9.1.7.3 DebugConnectionController . . . 85
9.1.7.4 DebugConnectionObject . . . 85
9.1.8 UserControls . . . 85
9.1.9 Debug Connection UserControl . . . 86
9.1.10 Robot Remote UserControl . . . 87
9.2 PotentialFields . . . 88
9.2.1 FieldView . . . 88
9.2.1.1 Changes . . . 88
9.2.1.2 Energyfield . . . 89
9.2.1.3 Directionfield . . . 89
9.2.2 Potentialfield . . . 90
9.2.2.1 User Interface . . . 90
9.2.2.2 Compile and send File . . . 90
9.3 BasicColorTableTool . . . 91
9.3.1 New functionalities . . . 91
9.3.2 New hotkey functions . . . 92
9.4 HorizonCalibrationTool . . . 94
9.4.1 Adding a Frame . . . 94
9.4.1.1 Tool Nr. 1: Horizon Based . . . 94
9.4.1.2 Tool Nr. 2: Reference Point Based . . . 95
9.4.2 Other Functions . . . 96
9.5 ImageViewer . . . 96
9.5.1 Motivation . . . 96
9.5.2 New functionalities in the ImageViewer . . . 97
9.5.2.1 Gradient color code . . . 97
9.5.2.2 Channel threshold calibration . . . 97
9.6 Motion Designer . . . 99
9.6.1 The Tool(s) . . . 99
9.6.1.1 The Main View . . . 99
9.6.1.2 Loading and Saving a ".mof"-File . . . 100
9.6.1.3 Editing a Special Action . . . 100
9.6.1.4 Execute Functions . . . 101
9.6.1.5 Additional Functions . . . 101
9.6.1.6 Window "EditMof" (only RC2) . . . 101
9.6.1.7 The Transition Editor (only RCXP) . . . 102
9.6.2 Button Interface . . . 102
9.6.3 Message Handling . . . 103

10 Challenges **105**
10.1 Passing Challenge . . . 105
10.1.1 Challenge Rules . . . 105
10.1.2 Problem Analysis . . . 106
10.1.3 Our Approach to the Challenge . . . 108

10.1.3.1 Role Assignment 108
10.1.3.2 Ball Handling 109
10.1.3.3 Communication 110
10.1.4 Evaluation of the Results 111
10.2 The New Goal Challenge 112
10.2.1 Challenge Rules 112
10.2.2 Problem Analysis 113
10.2.3 Our Approach to the Challenge 113
10.2.3.1 Detecting the Goal 114
10.2.3.2 Detecting the Free Part of Goal 117
10.2.3.3 Behavior 118
10.2.4 Evaluation of the Results 119
10.3 Open Challenge 119
10.3.1 Challenge Rules 120
10.3.2 Our Idea 120
10.3.3 Blind Dog 120
10.3.3.1 Player Model 120
10.3.3.2 Different Behaviors 121
10.3.4 Evaluation of the Results 122

11 Competitions **123**
11.1 RoboLudens 123
11.2 US Open 2006 124
11.3 RoboCup 2006 124

12 Conclusions **127**
12.1 ImageProcessor 127
12.2 Horizon Calibration 127
12.3 Ball Locator 128
12.4 Opponent Player Modelling 128
12.5 Behavior Learning 128
12.6 WalkingEngine 129
12.7 Special Actions 129
12.8 Tools 129
12.8.1 RobotControl XP 129
12.8.2 Potential fields 129
12.8.3 BasicColorTableTool 129
12.8.4 HorizonCalibrationTool 130
12.8.5 ImageViewer 130
12.8.6 Motion Designer 130
12.9 Challenges 130
12.9.1 Passing Challenge 130
12.9.2 New Goal Challenge 130

12.9.3 Open Challenge 130
12.10Competitions 131

A PotentialFields 133
A.1 Hints for RC2 133
A.2 Comments 133
A.3 Header 133
A.4 PotentialField 134
A.5 Structure of potentialfields-configuration 134
A.5.1 Standard-obstacle-objects 134
A.5.1.1 Functions 135
A.5.1.2 Appearances 136
A.5.1.3 Geometry 137
A.5.2 Object state symbols 137
A.5.3 Instances 137
A.5.3.1 Formations 138
A.5.4 Behavior 138
A.5.4.1 Motionfield 138
A.6 Additional features 139

B KickList 141
B.1 Kicks from grab 141
B.2 Kicks not from grab 142

Chapter 1

Introduction

The 4-Legged-League was first accomplished 1998 and although the rules changed every year a little bit, the games were always played with *AIBO*'s.

1.1 Playing Soccer with 4-Legged-Robots

The section deals with 4-Legged-Robots because the *AIBO*'s are 4-Legged-Robots and this leads to some scientific defiances.

1.1.1 Intentions

Playing soccer with robots is an initiative with the aim to foster research in robotics. The soccer game provides a set of scenarios with many real-life features. These features should help to solve problems that occur in real life environments which is claimed to be an important area of use in future robotics. By offering a standard complex problem, the strategies and methods for solving diverse sub-problems in the occurring scenarios are directly comparable. In different leagues slightly different aspects of research are focused. The 4-Legged-League offers not only a set of rules which imply the occurrence of certain complex challenges with legged movement. The rules also require the use of a completely standardized platform. This way development focuses nearly exclusively on the use of efficient software solutions. The last but not the least important factor that is in favour of the 4-Legged-League is its popularity among the outstanding spectators. This way robotics is consolidated into everyday life by having qualities of entertainment and therefore encouraging research and development.

1.1.2 Hardware

In opposite to other robot leagues, in the 4-Legged-League it is not allowed to modify the hardware. So all teams develop and play under equal conditions. The following list shows the hardware that is used:

- Dimensions: 180 (W) x 278 (H) x 319 (D) mm

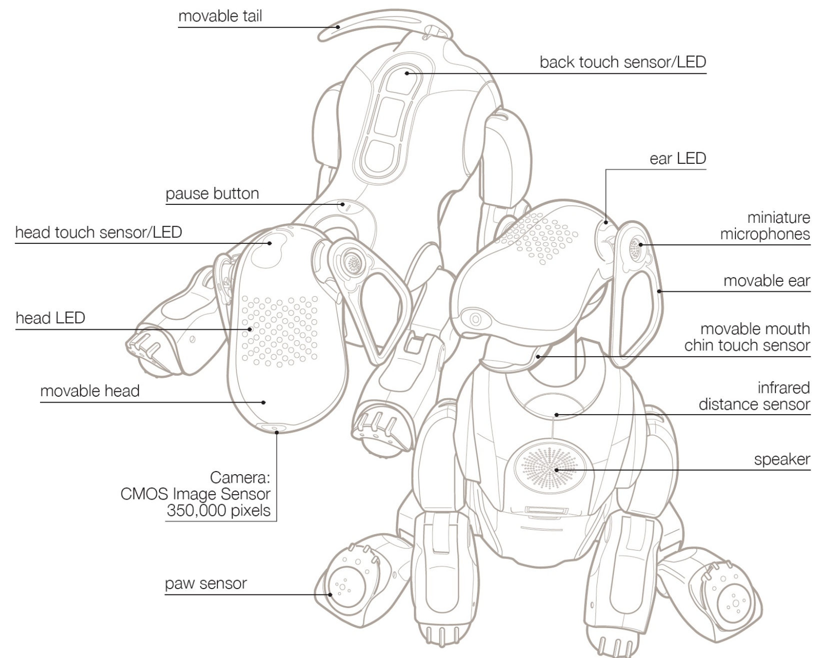


Figure 1.1: Sensors of *AIBO* ERS-7

- Weight: 1.65kg (including battery and memory stick)
- CPU: 64 bit RISC-processor with 576 MHz (MIPS IV)
- Memory: 64 MB SD-RAM
- Program storage media: *AIBO* memory stick
- WLAN Card: IEEE 802.11b (11 Mbps)
- Battery: Li-Io 7,2V 2200mAh

Internal sensors:

- Position sensors for all joints
- Acceleration sensor
- Vibration sensor
- Temperature sensor

External sensors and actuators: see Figure 1.1

- Camera: CMOS with 350.000 pixels, 30 frames per second
- Distance sensors: IR-sensors in chest and head

- Tactile sensors: on head, back, on each leg and under the mouth
- LED's: colored LED's on head and back
- Speaker: 500mW speaker connected to a 64 polyphonic sound chip
- Microphones: 2 microphones, one in each ear

The *AIBO* ERS-7 has 20 degrees of freedom overall.

- 4xLegs: 3 joints
- Head: 3 joints
- Tail: 2 joints
- Mouth: 1 joint
- 2xEars: 1 joint

With this hardware configuration the *AIBO* is able to run with up to 50 cm/s, to kick a ball, to communicate with other robots and to process his environment with up to 30 frames per second.

1.2 Innovations in 2006

During the last year we made improvements in many parts. The most important innovations are the following.

RobotControl XP For the debugging process new tools and modules were implemented. The main innovation is the new main tool **RobotControl XP**. For more information about the tool and its functionality take a look at chapter 9.1.

Players Detection For the game the players detection was improved. The detection is now co-operative and more dependable. For a closer look take chapter 5.

Behavior Learning Several behaviors can now be semi-automatically optimized. For this they are parameterized. You can find the description in chapter 6.

Ball Locator A new BallLocator approach using a kalman filter was implemented, modeling the position and velocity of the ball with higher accuracy and better reactivity than the previous GT2005 BallLocator, especially when the ball is moving. It started based on the GT2004 BallLocator ported into the current framework and since then was continuously improved. See chapter 4 for details.

Horizon Calibration It is now possible to correct the previously incorrectly calculated camera matrix and horizon semi-automatically with the help of several calibration parameters. A method to find these parameters is described in chapter 3.

Special Actions Special Actions can now be created directly on the robot. Either by connecting directly to the robot or even using the robot alone. Kneading the robot to get all parts in the right direction is also possible. For more information take a look at chapter 8.

Image Processor Now the ImageProcessor can choose between two color tables depending on the camera position. Detection of very close balls could be improved this way. Chapter 2 shows details.

Chapter 2

ImageProcessor

2.1 Ball Detection Improvements

The BallSpecialist that was used last year is already working well. But there are some inaccuracies in the detection if the ball is very close to the robot.

2.1.1 Problem Description and Motivation

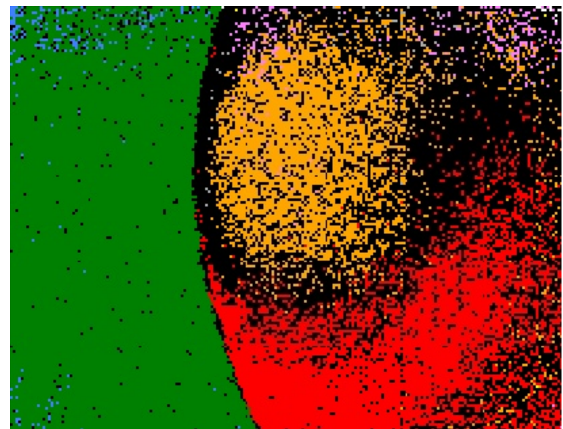
As already mentioned in the *GT2004 TeamReport*, one of the key problems in the RoboCup domain is to reach a high degree of robustness of the vision system against lighting variations. This is important, because we have to deal with unforeseen situations during competitions, such as additional shadows on the field as a result of the participation of a packed audience. To achieve this robustness we have been using generalized color tables since 2004 (For more details see [19]).
But there are also standard situations during the game, where the lighting conditions can differ, such as shadow casting due to other robots and even the robot's own shadows. Especially in the case when the robot looks at the ball and the ball is very close or even below the robot's head, then even a generalized color table cannot compensate these strong lighting changes.
Since the colors red, pink and yellow are located very close to the ball color orange in the color space, these lighting changes lead to a false classification and therefore to inaccurate ball recognition. Figure 2.1(a) shows the original camera image and figure 2.1(b) the color classified image. A standard color table was used to classify the image. The orange circle indicates the result of the BallSpecialist. We have a wrong calculation of the percept, because the orange classified patch is too small.

This inaccuracy is a big problem while handling the very close ball, because the success of grabbing, kicking and other actions with the ball depends on precise perceptions.

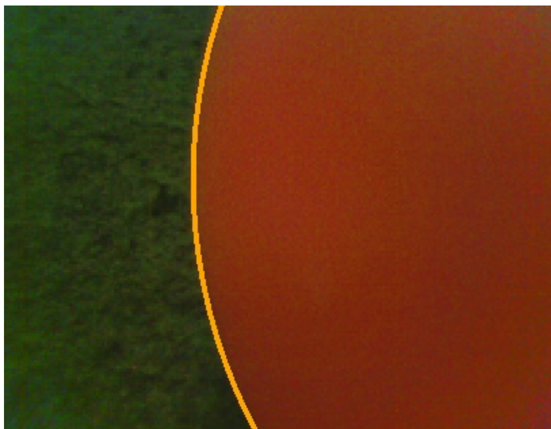
2.1.2 Solution Approach

Making color tables is always a kind of trade-off. Of course it is possible to create a color table that can perfectly segment the ball in a situation like it is shown in figure 2.1(b), but it also would

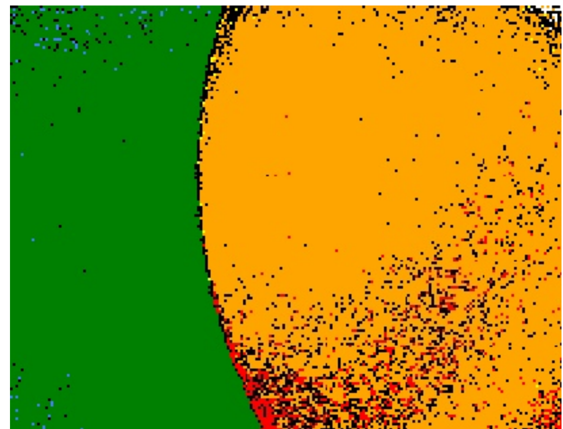
(a) A falsely recognized ball. The ball is too dark for a correct detection with an approach using only a single standard color table.



(b) A standard color table was used for the color classification. Only a small patch of orange was classified correctly.



(c) A correctly recognized ball. The additional *nearColorTable* was used.



(d) The *nearColorTable* classifies the close ball better.

Figure 2.1: Robot is looking down at a close ball. Comparison of the resulting percepts dependent on the used color table. The orange circle in (a) and (c) indicates the recognized ball.

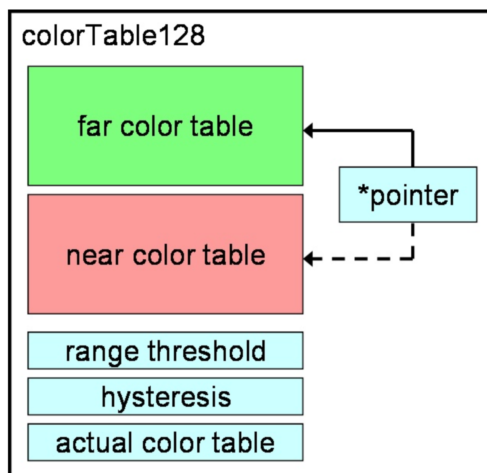


Figure 2.2: Elements of the *colorTable128* class. It consists of two color tables and additional information to decide situation dependent which one to use in the ImageProcessor.

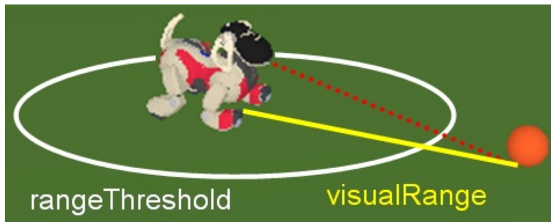
lead to recognizing balls in the red tricots of other robots, if the robot looks into the distance. So the key idea is to have an additional color table and to switch between the two color tables depending on the situation.

2.1.3 Description of the chosen solution

To reduce the amount of memory, the color table used on RoboCup 2005 is a sub-sampling of the YUV color space. Instead of 256 entries per channel, the *colorTable64* uses 4x4x4 cubes in the color space, so that only 64 entries per channel need to be stored on the robot (in the coltable.c64 file). Now, the new color table class *colorTable128* consists of two of those color tables, the *farColorTable* and the *nearColorTable*, the *rangeThreshold*, the *hysteresis* value, the variable *actualColorTable* that stores which color table is the active one, and a pointer to the active color table (see 2.1.3).

The *farColorTable* is the default color table. It is used when the robot looks into the distance. The *nearColorTable* is used when the robot looks down. To determine whether the robot is looking down, the so-called *visualRange* is calculated. The *visualRange* is the distance from the robot's position on the field to the intersection of the camera ray (ray from the projection center through the middle of the projection plane) and the field plane (see figure 2.1.3). This way, it is not necessary to consider all degrees of freedom of the head joints in order to determine whether the robot is looking down. If the robot looks into the distance, the *visualRange* value is large. Looking down results in a small *visualRange*.

If the *visualRange* is greater than the *rangeThreshold*, we assume that the robot is looking into the distance. (see figure 2.3(a)). The variable *actualColorTable* will then be set to the enum 'farColorTable' and the pointer is set to the beginning of the *farColorTable*. Accordingly, the *farColorTable* is used for image processing until the next frame. Otherwise we assume that the robot is looking down and the *nearColorTable* is chosen. (See figure 2.3(b)).



(a) If the *visualRange* (yellow) is greater than the *rangeThreshold* (radius of white circle), the *farColorTable* becomes active.

(b) Since the *visualRange* is smaller than the *rangeThreshold*, the *nearColorTable* is used.

Figure 2.3: The *visualRange* is calculated in every frame and compared to the *rangeThreshold* stored in the *colorTable128*.

The variable *actualColorTable* can be used within the ImageProcessor in order to query which color table is active. It also contains the information of the camera orientation, and this information could be used in the ImageProcessor, for instance, to decide not to start the BeaconSpecialist, because it is not possible to see beacons anyway when the robot is looking down (see [14] to get more information about the BeaconSpecialist). Within the ImageProcessor, the access to the active color table is realized by the pointer *colorClasses*. Depending on the *visualRange*, this pointer is set once every frame at the beginning of the ImageProcessor. Since the calculation of the *visualRange* is dependent on the camera matrix and thus on noisy sensor data, the *visualRange* fluctuates, even if the robot is standing still. Therefore a *hysteresis* value can be stored and considered in the calculation and the choice of the active color table.

2.1.4 Discussion of the solution's quality

The disadvantage of this solution is the fact that creating an additional color table is a time-consuming matter. But with some experience, the additional amount of time needed for creating two color tables can be compensated by their advantages. That is to say that the trade-off in creating the color tables is relaxed due to the separation of the situations where the robot is looking into the distance or looking down. Neither of the color tables needs to be as universally applicable as before with only one color table for all situations.

While making the *farColorTable*, we do not need to consider the ball when it is very close and therefore appears darker. Even the classification of pink is not needed when creating the *nearColorTable*, because beacons cannot be seen when the robot is looking down anyway. Additionally, some functions of the BasicColorTableTool can help to save time in the design of the *nearColorTable* (see BasicColorTableTool in the Tools section).

Finding a reasonable *rangeThreshold* is another requirement. A value around 250 mm is reasonable, but it can differ from robot to robot. This can be a problem, if the camera calibration and thus the calculation of the *visualRange* differs a lot between two robots. A switch between the two color tables could happen, for instance, too early, and the consequence could be false percepts. A well calibrated camera matrix or individual *rangeThreshold* values for every robot can account for additional improvements. In figure 2.1(d) the *nearColorTable* was used for color

classification. The orange circle in 2.1(c) shows the correctly detected ball. Thus, using two color tables brings more stability into the ball handling, if it is close to the robot. In a game, this could decide about making a goal or not.

2.2 Motion Compensation Calibration

The Motion Compensation has to correct the error that is caused by slowly sequentially reading from a CCD chip of a moving camera. Some preliminary tests revealed that a calibration of the existing system is necessary.

2.2.1 Description of the current system

The camera images are read sequentially from a CCD chip. If the camera is moving while taking an image, the first row of the image is recorded about 30 ms earlier than the last row. If the head, e. g., rotates with a speed of 180 degree/s, this results in an error of 5.4 degree for bearings on objects close to the upper image border. Therefore, the bearings have to be corrected. This correction is not performed as a preprocessing step for image processing, because it is a too time-consuming operation. Instead, the compensation is performed on the level of percepts, i. e. recognized flags, goals, edge points and the ball. An interpolation between the current and the previous camera positions (the so-called camera matrices) depending on the y image coordinate of the percept is used for the compensation. For more details see [19].

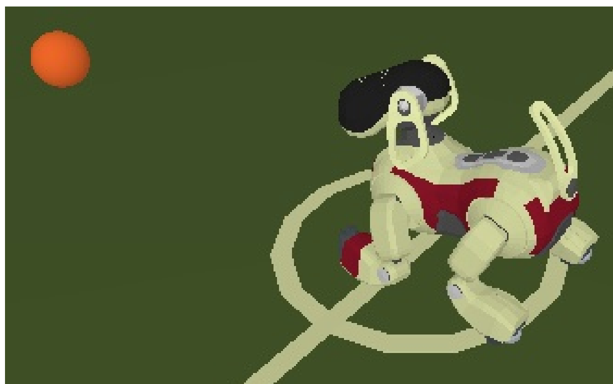
2.2.2 Motivation for the Calibration

To test the quality of the Motion Compensation, some measurements were necessary. The following test setup was used (see figure 2.4(a)):
The robot stands still on the field and only the head is moving on maximum speed (about 200 degrees per second) from left to right and back. A non moving ball is placed in different distances exactly in front of the robot. Every time the x-coordinate of the center of the recognized ball lies within a vertical stripe in the middle of the image, the rotation angle and the direction of the head movement (turning left or right) were recorded (see figure 2.4(b)).

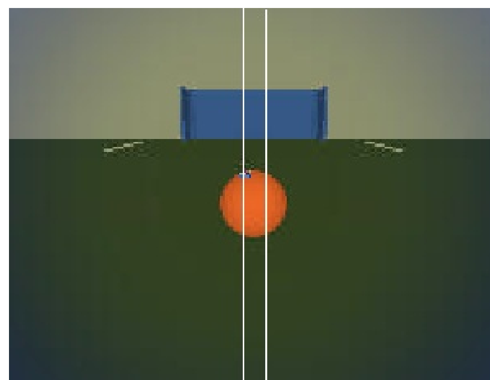
If the system works well, we would expect a distribution of the measured angles around zero degree (robot looks straight ahead). But tests show that there is a systematic error dependent on the direction of the movement. An erroneous correction of landmark percepts, for instance, could lead to inaccuracies especially in the self locator. This was the motivation to find the reason for the systematic error and to fix the problem.

2.2.3 Error and Calibration of the Motion Compensation

The *GT2004SensorDataProcessor* goes through the camera matrix buffer and chooses the camera matrix with the frame number that fits best. Both, the image and the choosen camera matrix, are then used in the ImageProcessor. But there are synchronization problems between the arrived



(a) The ball is exactly in front of the robot in a distance of 30-70 cm. The robot turnes the head fast from left to right and back.



(b) If the center of the detected ball lies within the stripe in the middle of the image (indicated by the two white lines) the head rotation angle will be recorded.

Figure 2.4: Test setup to check the quality of the Motion Compensation

image and the corresponding camera matrix. The measurement of the rotation angle around the z-axis shows that all angles measured in a head movement to the right side are positive. In a head movement to the left side, the angles are negative. That means, if the ball is seen in the center of the image in a right head movement, the calculated ball position is shifted too much to the left side, in a left movement to the right side. Figure 2.2.3 illustrates this systematic error.

The explanation for the measured error is that the camera matrix is older than the image. Thus, one part of the calibration of the Motion Compensation is to use a newer camera matrix. This is possible because there are always four camera matrices calculated between two images. Additionally, the interpolation factor (see description of the current system) was calibrated by changing the factor a little and checking the resulting accuracy by the described test method.
The standard deviation was used as a quality criterion. The second criterion was the distribution of the head movement directions after sorting the recorded data by the corresponding head angle value. The systematic error still exists when the directions are sorted (for instance first all movements to the left and than all movements to the right). The more the movement directions are mixed up, the less the systematic error exists.

2.2.4 Results of the Calibration

In more than 5000 measurements on different robots, the calibrated Motion Compensation was compared to the old version. The standard deviation went down to about 1 degree, which is less than half of the old standard deviation. Figure 2.2.4 shows the distribution of the head movement directions where the corresponding measured angles are sorted in ascending order. No systematic error could be found any more.

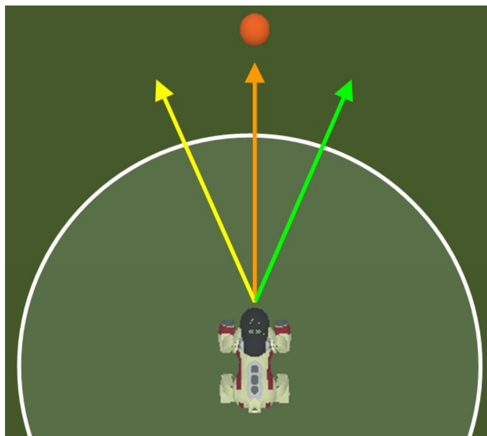


Figure 2.5: Systematic error in the Motion Compensation: The orange arrow shows where the ball is seen in the image. The yellow arrow shows the calculated position of the ball after the Motion Compensation during a head movement to the right side, the green arrow the calculated position during a head movement to the left side.

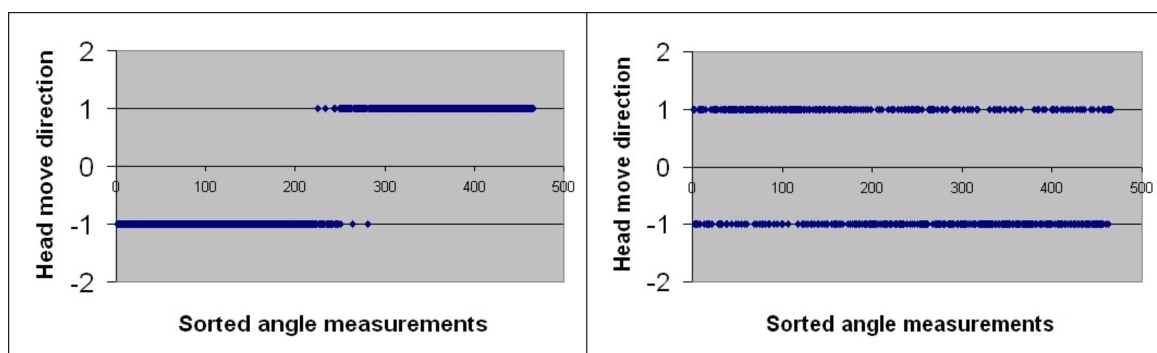


Figure 2.6: The diagrams show the distribution of the head movement directions when the corresponding measured angles are sortet in ascending order. Left diagram: old Motion Compensation; right diagram: calibrated Motion Compensation. The value 1 stands for a head movement to the left, -1 for a head movement to the right. There is no systematic error any more.

Chapter 3

Horizon Calibration

The horizon line in an image plays an important role in our ImageProcessor: The image is scanned for important features along scan lines which are aligned parallel to the horizon. Furthermore certain features are only searched for at their expected position relative to the horizon, e. g. the baseline of the goals below, and the landmarks above the horizon [14].

3.1 Problem Depiction

The horizon line itself is derived from the camera matrix which is calculated for every incoming image from the robot's joint values. It represents a homogeneous transformation from the camera frame to the base frame. While also the leg joint values are used for calculation, the most important joints are the three head joints, namely the *neck-tilt-joint*, the *head-tilt-joint* and the *head-pan-joint*. Unfortunately the potentiometers in the joints don't measure the positions very accurately, which results in an erroneous camera matrix and horizon line (Figure 3.1(a)).
Measurements have shown, that the errors in the joint values are of linear nature (Figure 3.2), therefore we have introduced the following calibration values to correct the camera matrix, where special regard is paid to the head joints: An offset is added to the overall body roll and body tilt, calculated by the leg sensors, while each of the three head joints are calibrated by performing a multiplication with a factor and adding an offset. This gives us a sum of eight calibration parameters, which need to be determined for each *AIBO*.

3.2 Different Approaches

In the past it was tried to find the above mentioned parameters by hand, but this proved to be difficult, since more than one head position needs to be considered for this, and the parameters are interdependent. In practice only a rough calibration with one or two parameters was done.
This section describes some methods that were experimented with during the last year. All methods have in common, that the calibration is achieved via reference values in the real world. Attempts have been made to use the real horizon of an image in order to calibrate the falsely calculated horizon and the camera matrix, or to use reference points in a calibration environment,

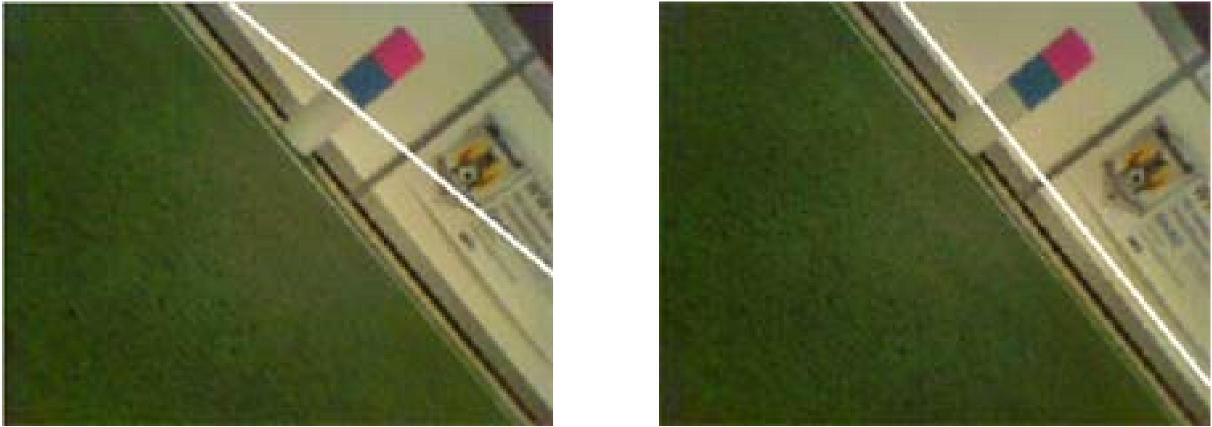
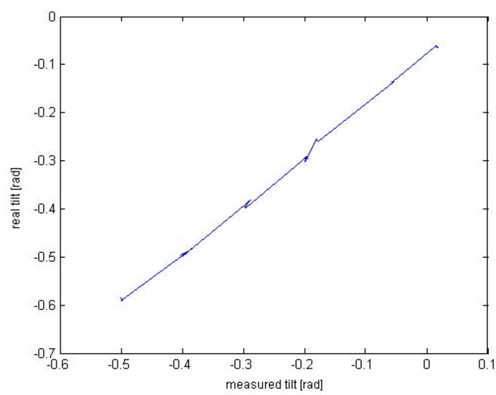
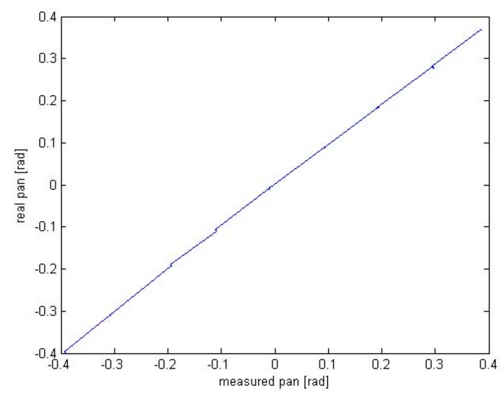


Figure 3.1: The horizon as calculated by the robot before (a) and after (b) the calibration



(a) Neck Tilt Joint



(b) Head Pan Joint

Figure 3.2: Measurement of the calculated vs. the real joint values

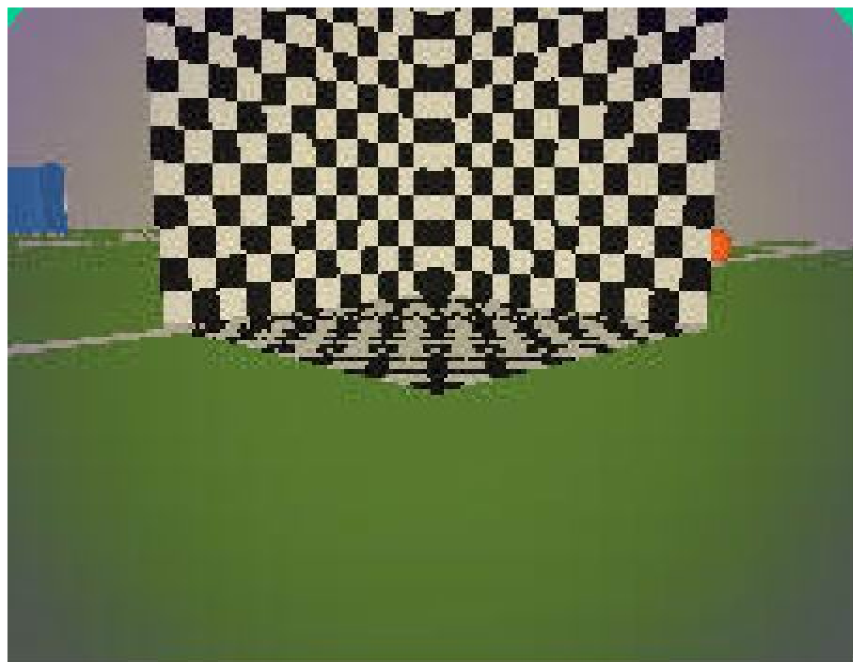


Figure 3.3: DLT: Reference points

in order to calculate the calibration values. As a means to find the eight values, an evolutionary algorithm is used. In the beginning, each parameter starts with a neutral value, i. e. the offsets are 0 and the factors are 1. In each iteration of the algorithm the parameters are randomly mutated, and a fitness function, which is different for each approach (cf. the sections below), evaluates whether the result is better than the previous evolution.

3.2.1 DLT-Method

One Approach to calculate the camera matrix, is the Direct-Linear-Transformation-Method [9] (DLT-Method). The DLT uses reference points (Figure: 3.3) in the image and a linear world model to calculate position and orientation relative to the reference points. In the easiest implementation, six points are enough to calculate all six parameters, but this assumes undistorted and corrected images, or leads to less exact values. With more than six reference points, it's possible to calculate the parameters with least-square-error or to add more equations and consider also some non-linearity in images. This leads to a non-linear modified DLT [10] where at least eight control points are needed.
For this method a pseudo-inverse of at least an 11x12 matrix must be computed, where the number of rows is twice that of control points and the number of columns raises to 16 if a more accurate DLT is used. For the modified DLT, the inverse has to be computed in each iteration.

3.2.2 Automatic Horizon Detection

Another approach aims at using the actual horizon as a reference for finding the eight calibration values. The basic idea of this is to take several images in different head positions. For each of these images, the robot's joint values and the y-position of the actual horizon at the left and right side of the image are stored and taken as the input for the evolutionary algorithm. As mentioned above, the algorithm then creates a new set of calibration parameters for each iteration. The fitness of one evolution is hereby determined as the standard deviation between the position of the real horizon and the horizon calculated by the calibrated joint values for the taken image frames:
For each frame that has been stored before, the camera matrix is calculated with the stored joint values, which are corrected by the current calibration values. From this we can determine the horizon line as it would be calculated by the robot with the given calibration parameters, using

$$h_l = \frac{r_{33}v_0 + r_{31}f + r_{32}u_0}{r_{33}} \tag{3.1}$$

$$h_r = \frac{r_{33}v_0 + r_{31}f - r_{32}u_0}{r_{33}}. \tag{3.2}$$

v_o and u_0 define the optical center of the camera, f the focal length. The camera matrix is given by $R = (r_{ij})$.

Now the horizon line calculated like this is compared to the real horizon, that is stored with the joint values, and the sum of the deviation of all frames calculates the fitness function of this parameter set:

$$FITNESS = \sum(|y_{l,i} - h_{l,i}| + |y_{r,i} - h_{r,i}|)^2 \tag{3.3}$$

with $y_{l,i}$ and $y_{r,i}$ being the left and right side coordinates of the calculated horizon, and $h_{l,i}$ and $h_{r,i}$ those of the real one.
The most interesting aspect of this, though, is the question of how to acquire the 'real' horizon. As mentioned in [13] the line of the horizon is formed by the vanishing directions of the ground plane. That means that we need to detect the so called vanishing points of that plane, and the horizon line is the line that intersects these two points. One method to detect vanishing points is described in [12]. The drawback of this method is, that it can be only used, if there is just one dominant vanishing point in the image, like if one looks along a straight road. In order to find the horizon line, it is necessary however, to find two of such points.
Therefore this method has been extended, in order to accommodate our means. At first the entire image will be scanned, and all image points with a gradient greater than a certain threshold will be converted to polar space (ρ, Θ). In order to detect two different vanishing points with the method from [12], the set of polar lines needs to be divided into two sets, one for each vanishing point. This is done with a 2-means clustering algorithm, which will operate in hough space. As initial centers for the two clusters, the left and right side coordinates of the calculated horizon could be used. Each point in polar space is then assigned to the cluster with the distance

$$d = |\rho_i - (x\ cos\Theta_i + y\ sin\Theta_i)| \tag{3.4}$$

being the smallest, where x and y are the image coordinates of the cluster's centers. After each point has been assigned to one of the two clusters, the method in [12] is used to find the vanishing points defined by the points in each of the two clusters. The hereby calculated vanishing points are then used as the new centers of the two clusters, and the whole procedure starts again. As with every clustering algorithm this is repeated until the centers of the clusters, i. e. the vanishing points, don't move anymore. These two found points now define the horizon line of the image.

3.2.3 Manual Calibration

3.2.3.1 Horizon-Based Method

As the automatic horizon detection method mentioned above proved not to be as stable as expected (cf. 3.3), the automatic part has been reduced to a manual part. While the final calculation of the evolutionary algorithm still works as before, the way the real horizon is detected has changed. In this method it is done manually by the user, with the help of the HorizonCalibrationTool in *RobotControl*. One characteristic that is used to find the horizon is, that all points in the world, that lie at the height of the camera, i. e. at the horizon, project to the horizon line in the image, independent of the actual distance to the camera. This means, that if a line in the real world lies at the height of the camera, its projection on the image plane doesn't move if the camera is moved back and forth.
An easy way to use this in finding the horizon for our robots, is to use a computer screen and any software that is capable of drawing and moving straight lines. The robot is placed before the screen and looks at a line. The robot is then moved back and forth to observe the line in the camera-image. If the line is moving in the image, then its height on the screen will be adjusted accordingly and the robot is moved again, until the line on the screen is in such a position, that the line in the image doesn't move anymore. This would then be the horizon line, and it can be marked by the user.

3.2.3.2 Reference-Point-Based Method

With the camera matrix it is possible to transform world coordinates into image coordinates and vice versa. Thus it is also possible to correct the camera matrix with a set of image-world-reference-points $(x, y, z), (u, v)$. This works as follows: The robot is placed in a calibration environment (Figure 3.4) with markings at some specific world points. With the help of the HorizonCalibrationTool in *RobotControl*, these world-coordinates are mapped to the respective image points. This is done for several points with different head positions. The coordinate pairs are then used as input to the evolutionary algorithm together with the robot's joint values, which are gathered automatically by the tool.
The fitness function for the algorithm is now given by the deviation between the calculated image points and the real positions given by the user.

$$FITNESS = (\sum \sqrt{(u_{i,real} - u_i)^2 + (v_{i,real} - v_i)^2})/n \tag{3.5}$$

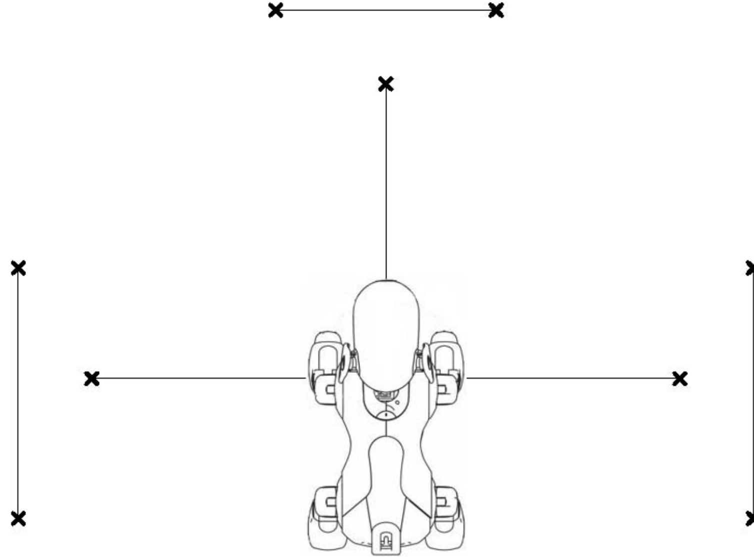


Figure 3.4: Calibration Environment

where $u_{i,real}$ and $v_{i,real}$ are the image coordinates specified by the user, and u_i and v_i are the image points calculated for the current set of calibration values using

$$(\tilde{x}_i, \tilde{y}_i, \tilde{z}_i)^T = R^{-1}\,(x_i, y_i, z_i)^T \tag{3.6}$$

$$u_i = u_0 - \tilde{y}_i \frac{f}{\tilde{x}_i} \tag{3.7}$$

$$v_i = v_0 - \tilde{z}_i \frac{f}{\tilde{x}_i}. \tag{3.8}$$

(u_0, v_0) is the optical center of the camera, f is the focal length. $(x, y, z)^T$ are the world coordinates of the regarded point, R is the current camera matrix, which is corrected by the current set of calibration values.

3.3 Evaluation and Achieved Result

The solution we tried to implement first was the DLT-Method. Unfortunately, we had to abandon this idea after a while. For a still unknown reason, the results that this method provided were utterly wrong.

As a consequence this idea was discarded and we moved on to the next idea, namely the horizon-based calibration. The weak part in this approach was that of the automatic horizon detection. When the two vanishing points that were to be found, were located closer to the center of the image, they were detected correctly, but the further they were away from the center, the more

imprecise did the detection get. This can be explained the following way: The vanishing points are detected through the gradient of the horizontal lines on the checker board, which intersect at this point in the image. If the point lies close to the center of the image, the method is more robust to errors, and all the lines intersect in pretty much the same point in the image. If the vanishing point lies further away from the center, even outside of the actual image, then this point of intersection becomes more blurred. Here a small error in the direction of the gradients, can result in a large error in the detection of the vanishing point. One important factor here, is the low resolution of the image: because of the resolution, the determination of the direction of the image gradients is quite error prone. That's why, while in principle this method would work, it cannot be used on the *AIBO* robots in the way presented here. It would be too imprecise.
Though the automatic detection didn't quite work, the actual calculations done by this method (i. e. the evolutionary algorithm) could still be used. The only thing was, that the horizon would have to be found manually. Doing this was quite tedious work, but the result was very satisfactory. The position of the horizon could be corrected for every head position (Figure 3.1). Admittedly there were still some strange errors: The *catch-ball-high* head control mode, and any similar head positions, appear to be quite instable regarding sensor readings. If the robot first looks straight ahead and then moves to the mentioned head position, the horizon may be calibrated correctly, but if the robot first looks right or left, and then to the *catch-ball-high* position the horizon is either too high or too low. This problem couldn't be fixed so far, and would probably involve a calibration, based not only on the current head position, but also on head movement.
The last method described earlier was introduced to avoid the manual finding of the horizon. Despite from being more comfortable, the achieved results were more or less the same. The above mentioned error was also present when using this method.

Chapter 4

Ball Locator

4.1 Goals of Ball Modeling

The basic idea of Ball Modeling is to filter the percepts from the ImageProcessor to create a stable and accurate model of the position and velocity of the ball. The robot should have a reliable 'idea' about where the ball is and where it is moving, even when it was not seen for a while due to obstructed vision or the robot looking in another direction, so it will be easier to find again. One primary goal is to filter the sensor noise from the ImageProcessor, that depends on several factors as for instance the distance to the BallPercept, the movement of the head or even wrong percepts in other yellow-orange-colored objects in- or outside the field, and this way creates sometimes 'jumpy' percepts of a ball that is in real not moving at all. Another goal is to be able to 'guess' with a certain reliability where the ball is, when it was not seen for a while, and also to predict from its previous movement, where it will move next.

4.2 Possible Solution Attempts

Two major attempts to achieve these goals are the so-called Particle Filters and Kalman Filters. Both are 'probabilistic' filters representing a ball's state by a so-called state vector, containing the necessary information about the ball like position or velocity. They mainly work in two steps, a Time-Update step considering changes of the ball state over time, also negating the effect of robot movement to the relative position of the ball and continuing loss of reliability, and a Measurement-Update step incorporating the latest BallPercept and increasing reliability.

4.2.1 Particle Filter

Particle Filters [8] use stochastic methods from the Monte-Carlo-Scheme [4] and model a certain number of 'possible' state vectors, now called Particles. In the Time-Update step all these are changed simultaneously. The Measurement-Update step now 'updates' each particle according to its distance to the latest BallPercept in a way the Particles begin forming a 'cloud' around it. After both steps finally the average state vector is calculated by means of clustering algorithms.

4.2.2 Kalman Filter

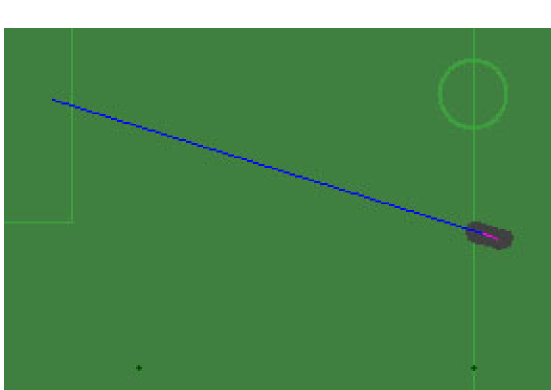
Kalman Filters [20] assume the noise of the system as gaussian noise and the movement as linear. They work with only one state vector, the Mean, and its gaussian uncertainty represented by the so-called State Covariance Matrix. During the Time-Update step, now called Prediction step, the state vector is multiplied by a so-called State Transition Matrix modeling its linear change, assuming it is moving on linearly, and the state covariance matrix is changed according to the process noise. Thus the current ball state is 'predicted' according to its last movement. In the Measurement-Update step this prediction then gets corrected incorporating the latest BallPercept, taking into account its reliability by the so-called Measurement Covariance Matrix and the reliability of the predicted state by the Process Covariance Matrix, calculating the new Mean state and its State Covariance Matrix. The values in the measurement covariance matrix for example define how much the filter 'trusts' the information from the BallPercept, higher values mean lesser trust, because the values say by how much uncertainty the according measurements are affected. This is similar for the Process covariance matrix as well as for the State covariance matrix itself.

4.3 Chosen Solution: *MSH2006BallLocatorKalman*

Considering several papers [18][5] about Ball Modeling from other teams, sometimes also from other leagues, it was decided to try a Kalman Filter approach. In 2004 there was already tried such a solution, but because in 2005 there was implemented a solution based on particle filtering, which achieved better results, this approach was no longer in the framework. Also the ball model the ball locators are working on and the framework itself was changed significantly and the approach from 2004 was not compatible anymore. So for simplicity's sake to get started the *GT2004BallLocator* [14] was ported into the current framework, to be able to see a ready implemented Kalman Filter at work.

4.3.1 Basic Concept

The first step then was to implement a new BallLocator, using the significant parts from the ported *GT2004BallLocator* in the context of the framework of the current *GT2005BallLocator* [15], on the one hand to make use of the latest improvements in percept processing and on the other hand to be sure to set all current Ball Symbols properly. To keep work simple and to be able to compare both approaches, the old 'intuitive' methods from the *GT2004BallLocator* to make the Measurement Covariance Matrix dependent on factors like "distance to the percept" and "panning velocity of the head" were still used in their original form. The next step then was to implement an own system of covariance matrix adaption dependent on the several factors for uncertainty mentioned earlier. Because they only affect the certainty of measurements, it was decided to adapt only the Measurement Covariance Matrix, and for simplicity's sake this was done taking into account only the two factors "distance to the percept" and "panning velocity". Also the measurements taken later proved that other factors don't affect it significantly.



(a) Ball seen, very low uncertainty, small spread

(b) Ball not seen for a while, high uncertainty, large spread; Yellow crossing lines: Standard Deviations of the State Covariance Matrix along the axes

Figure 4.1: Dummy particles shot around the ball position as interface to the TeamBallLocator

Because the TeamBallLocator [15] was still based on a particle filter and needed a certain amount of representative particles from the BallLocator as an interface, it was also necessary to implement a function to create 'dummy' particles representing the currently modeled ball position and its reliability. The solution was to shoot a certain amount of particles randomly in gaussian distribution around the modeled ball position, using the standard deviations calculated from the state covariance matrix (def.: std. deviation = variance squared) to determine the spread, as seen in figure 4.1.
The standard deviations are also used to determine the reliability of the modeled position and velocity, by comparing their value against a certain threshold. Values above that, mean a reliability of 0, standard deviations of 0 mean a reliability of 1.

4.3.2 Covariance Matrices by Measurements

Covariance Matrices per definition consist of the Variances on the main diagonal and in case of interdependence also Covariances on the according elements, but in our case it is too difficult to measure interdependence between x- and y-components or between position and velocity, so it is only necessary to determine the four variances for the main diagonal, letting the other elements of the matrix be 0.

$$\begin{pmatrix} \sigma_{px}^2 & 0 & 0 & 0 \\ 0 & \sigma_{py}^2 & 0 & 0 \\ 0 & 0 & \sigma_{vx}^2 & 0 \\ 0 & 0 & 0 & \sigma_{vy}^2 \end{pmatrix}$$

Variance (σ^2) now is per definition the average of the squared errors (δ^2).

$$\sigma^2 = \left(\sum_{i=1}^{n} \delta_i^2\right) / n$$

The last necessary step is to rotate the final matrix according to the angle to the BallPercept as seen in figure 4.2.

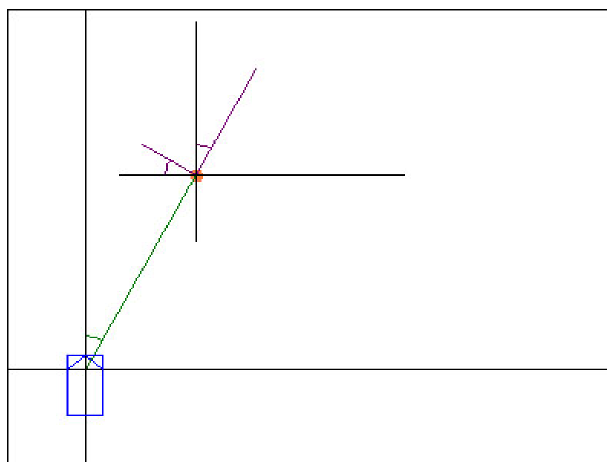


Figure 4.2: Rotated Covariance Matrix (Variances represented by purple axes)

The basic idea to determine a function for these variances now was to approximate them from a series of measurements. For this purpose own measurement behaviors were created, making a series of measurements under certain conditions and calculating the variances together with the aforementioned factors. Two kinds of behavior were created, one to estimate the dependence of the error from the distance to the ball, where the robot walks towards the ball and stops regularly to take the measurements, and the other to estimate the dependence of the error from the panning velocity, where the robot stands still and moves the head with a given velocity while it takes the measurements. The following figures display drawings of several measurements (represented by lines with dots) together with different approximation attempts (represented by plain curves).

4.3.2.1 Position Variances

In figure 4.3 and 4.4 it can be seen that for example, the maximum measured x-position variance would be about $0.06m^2$ for a distance of about $4m$ to the ball. It can also be seen that the influence of the distance on the y-position variance is not that significant, for example it is only about $0.0002m^2$ at about $3.5m$ distance. When the distance came below $300mm$ the ball was too close to the robot to keep it fixed in the view of the camera, the robot started rapidly moving its head, thus producing those unusual high variances. Because this is a problem of the control of head motion and the recognition of the ball in the ImageProcessor, which cannot be solved in the BallLocator, this kind of 'noise' was just ignored and in first attempts the approximative curve was flipped vertically at $300mm$. Later some extra 'improvised' approximations for distances

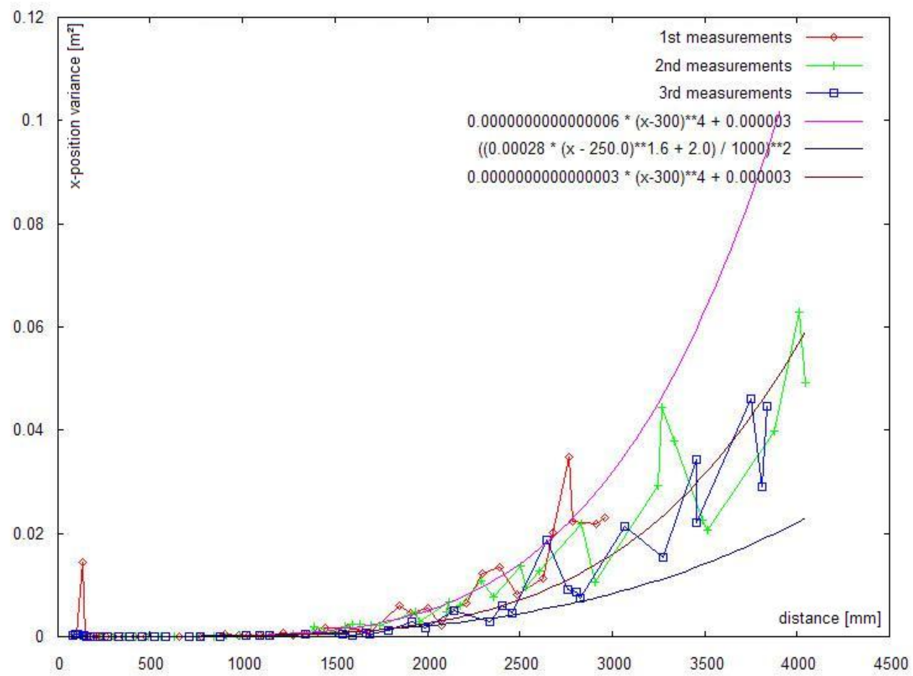


Figure 4.3: x-Position Variance dependent on Distance

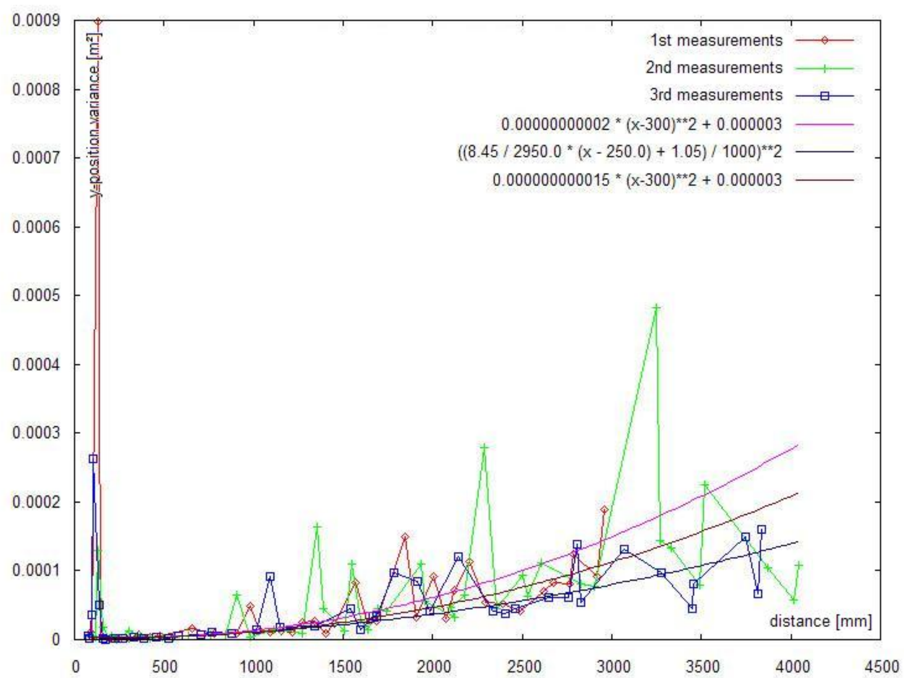
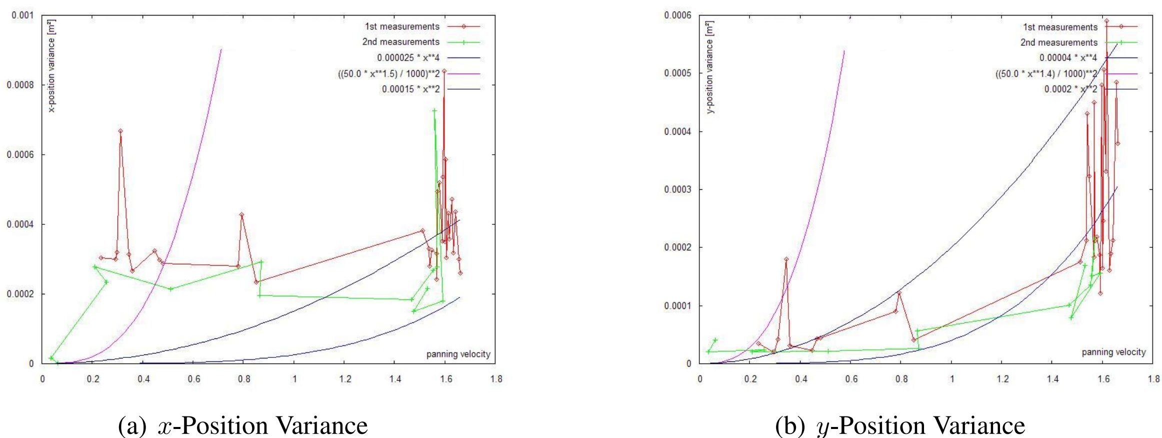


Figure 4.4: y-Position Variance dependent on Distance

below $300mm$ were created to further filter out that noise and create extra stability. Also for the very first set of approximations leading to the first accurate and stable *Kalman BallLocator* the variance was still measured in $[mm]$ and because of an initial lack of understanding was not variance (per definition) at all. Instead it was calculated as the average error and in the code later got squared. So it was necessary to divide these approximation functions by 1000.0 and then square them to be able to compare them against the other functions with correct variance. Surprisingly it was very accurate and stable and it took a large number of alternative attempts with correct variance (per definition) to create a solution that was better in the end. But as it can be seen in figures 4.5(a) and 4.5(b) this first 'wrong' approach is quickly high above the measured variances dependent on panning velocity, thus creating a very stable model for when the robot moves its head. It can also be seen that the measurement of the panning velocity itself is very noisy, because inspite of being raised almost linearly the velocity in the plots seems to be only jumping around. But with help of the approximations one may get the idea that this is indeed just a 'very noisy kind of' curve. Also it was surprising to see only such low variances in these measurements, a maximum variance of about $0.0008m^2$ for x-position and $0.0006m^2$ for y-position is not that significant, considering panning velocity as the factor most influencing stability by observations.



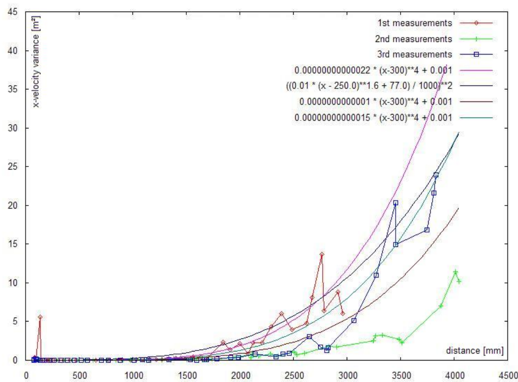
(a) x-Position Variance (b) y-Position Variance

Figure 4.5: Position Variances dependent on Panning Velocity

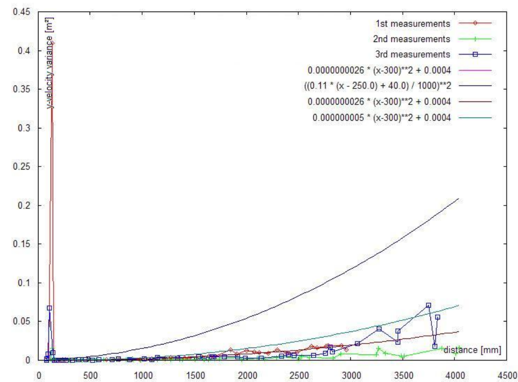
4.3.2.2 Velocity Variances

In the very first attempts the same approximations were used for Position and Velocity. Soon it was found out, that this was the cause for the failed attempts to achieve results similar to the first approach (the 'wrong' one) with now correct variances. The velocity of a ball not moving at all was quickly measured as very high when the robot was moving its head, thus the model became very 'jumpy' and instable in this situation. This was previously prevented (without knowledge) by the very high variance for panning velocity, that was by this time also used for the velocity measurements. Now with approximations fitting the curve of the measurements for position variance it became necessary to use different variance functions for position and velocity and the

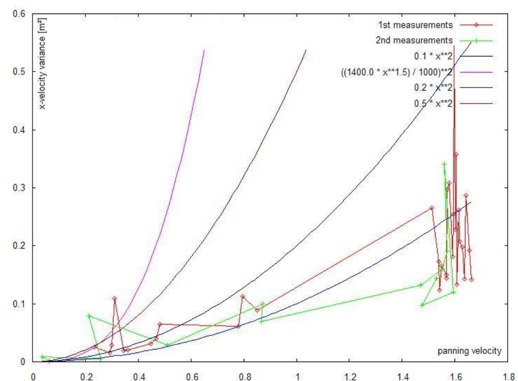
measurements done for this also proved it.
In figures 4.6 it can clearly be seen now that the effect of the distance or the panning velocity on the velocity measurements is indeed much greater than on the position measurements (e. g. about $20m^2/s^2$ for x-velocity, $0.05m^2/s^2$ for y-velocity dependent on distance and $0.5m^2/s^2$ for x-velocity, $0.15m^2/s^2$ for y-velocity dependent on panning velocity). Also the curve from the 'wrong' approach is almost always above the measurements, that is the cause for why it was so stable all the time.



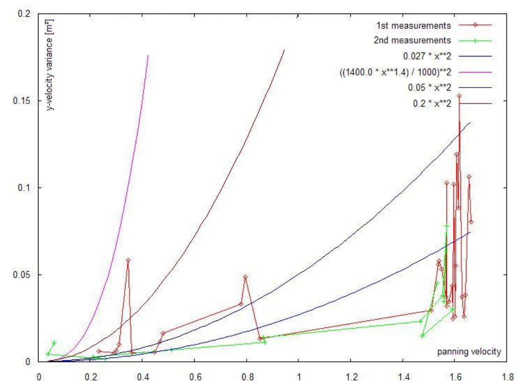
(a) x-Velocity dependent on Distance



(b) y-Velocity dependent on Distance



(c) x-Velocity dependent on Panning Velocity



(d) y-Velocity dependent on Panning Velocity

Figure 4.6: Velocity Variances

4.3.2.3 Conclusion

Now considering all those measurements, approximation approaches and resulting model behavior, the goal is to have very reliable measurements, to calculate mostly fitting highly accurate approximations and keeping in mind the following observed guidelines. Curves below measurements result in higher reactivity, while curves above them result in more stability. For position variances dependent on distance it is a good idea to fit the measurements as accurately as possible, while dependent on panning velocity it would be better to 'touch the peaks' to

better filter this 'jumping'. And because velocity measurement itself is very noisy and has the greatest impact on stability, it is better to let the curves for velocity variances be a bit above the measurements for less over-reacting and more stability.

4.3.3 Additional Improvements

Besides the continuous improvements to variance approximations there have also been done additional improvements to increase accuracy and stability. Especially after no significantly better results were achieved by further variance approximations, this became the most important part of further improvements.

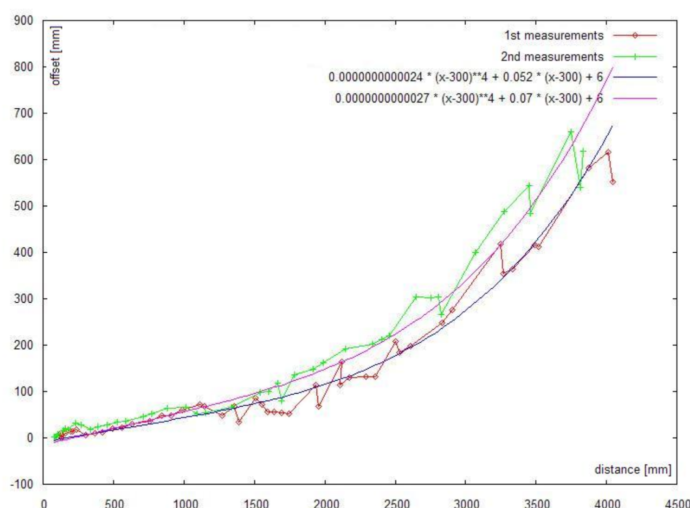


Figure 4.7: Offset between Percepts and Ceiling Cam distance measurements

4.3.3.1 Distance Offset

The first additional improvement at all (although it was the longest period of development parallel to the variances until it finally achieved useful results) was the elimination of the offset in distance measurement between the BallPercept by the robot's camera and the ball position measured by the ceiling cam, realized already throughout the very first measurements. The robot tends to always locate the BallPercept far behind where it actually is, increasing by distance (even with correctly calibrated camera matrix and horizon and on new robots with best visual quality). Figure 4.7 shows this increasing offset dependent on the distance perceived by the robot. For example when the robot perceives the ball in $1.5m$ distance, it is actually rather in $1.4m$. So a distance error correction function was created, again using approximations from measurements. The lowest 'error' occured at about $300mm$, which was, not surprisingly, the same base distance as for the variance measurements, and below that of course the robot again began constantly moving its head producing the same noise in measurements, so according to previous discoveries

for distances below that, again a separate approximation was created. The solution now was to simply subtract the value of the approximated function from the percepted distance and to recalculate the according ball position before processing the percept with the filter.

4.3.3.2 Non-Linearities

One of the main drawbacks of the kalman filter is obviously its assumption of a linear change of state. The movement of the ball is not linear at all, at least it constantly slows down depending on the friction coefficient of the carpet. This is taken into account in another additional improvement. The linear transition of the state is done by multiplying the state vector by the state transition matrix, previously the unity matrix adapted by time difference (dt) each step. So the velocity is kept constant and multiplied by time difference is added to the position, to create the linear movement of the ball.

$$\begin{pmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This matrix is recalculated each prediction or measurement update step, incorporating a constant slowdown (l: absolute value of velocity, nl: absolute value slowed down, μ: coefficient of friction): $nl = l - (\mu * 9.81 * dt)$

$$\begin{pmatrix} 1 & 0 & (nl/l) * dt & 0 \\ 0 & 1 & 0 & (nl/l) * dt \\ 0 & 0 & (nl/l) & 0 \\ 0 & 0 & 0 & (nl/l) \end{pmatrix}$$

This way the velocity is constantly slowing down depending on the friction coefficient of the carpet, also affecting the calculation of the new ball position after “pseudo-linear” movement. Because the effect of this on the model is not visible when the ball is constantly seen, but only when ‘rolling out’ while not seen, it is not that important to determine the ‘exact’ coefficient of friction for a carpet. It is sufficient to ‘guess’ a working coefficient by experiment (e. g. shooting the ball past the robot outside its field of view and compare the positions where it stops in the model and in reality, trying several coefficients, until a satisfactory value is found).

4.3.3.3 Velocity Filtering

Due to the extremely noisy measurement of velocity, certain techniques were developed to filter it before it is used as input to the filter. No velocities above $1.5m/s$ are allowed and all velocities are clamped to this range. The measured velocity gets averaged over 5 frames using a rotating buffer, always substituting the oldest measurement by the latest. Velocities are only added to

the buffer, when the ball was seen continuously for a certain amount of frames and the panning velocity is below a certain threshold, otherwise the buffer is emptied completely and the last averaged value, now just slowed down by time according to friction, is used instead. Also when the distance to the BallPercept exceeds a certain threshold, the distance from the percept to the modeled position is used to determine the measured velocity instead of the distance to the last percept.
One of the main problems of a linear change in modeled velocity was, that especially when the direction of the ball's movement changed, the modeled velocity kept pointing in the old direction constantly slowing down then starting to turn in the right direction. An attempt to eliminate this problem was to interpolate between the modeled direction and the direction extrapolated from the last movement (latest attempt was 60% from movement direction and 40% from modeled. This could perhaps be made dependent on the angle between the two directions instead to make it react faster).

4.3.3.4 Experimental Forecast Method

The only improvement never really used was the experimental forecast method, able to predict where the ball will be at a time in the future. It simply used the time-update method of the prediction step (assuming the ball will move on in the same direction as before, constantly slowing down) without changing the state vector, but instead creating a copy of it, applying the time-update step for a time in the future and returning the new state vector to be used instead of the model this time. In first experiments the amount of seconds the predicted time lied in the future was determined by the distance to the ball, allowing faster reaction when the ball moved towards the robot, but also keeping reliability when it was near, especially when dribbled. But because it uses the prediction step method, which only "pseudo-linearly" moves on and constantly slows down, its reliability is too low to make practical use of it yet.

4.4 Evaluation of Results

To compare the results from different approaches a "*ComboBallLocator*" was used, running two BallLocators in parallel and comparing each against ground truth from the ceiling cam. Statistical values as the error in position and the error in absolute value and direction of the velocity can be averaged over a certain number of frames, as well as for each value the average percentage of frames *BallLocator 1* was better than *BallLocator 2*. To get representative results it was necessary to get data for certain different situations, e. g. ball lying at different positions with varying angle and distance to the robot, especially very far and very close, ball shot 'by hand' into certain directions and finally shooting the ball around measuring over a couple of minutes and thus simulating a game situation, and to compare stability doing similar tests with constantly moving head searching for landmarks. At first, after the *GT2004BallLocator* was ported into the current framework, it was too unstable to even consider a comparison, but soon after the first improvements were done creating the first own solution, it already achieved much better results than the *GT2005BallLocator*.

One thing not measured by the *ComboBallLocator* was the reaction on a shot ball, while the *Kalman BallLocator* started 'tracking' the ball a short time after it was shot, constantly modeling the current location of the ball "pseudo-linearly" [section 4.3.3.2] moving and corrected by new percepts, the *GT2005BallLocator* started spreading the particles around the modeled ball position which was at that time not moving at all. When the ball slowed down to a stop, the *Kalman BallLocator* modeled the location already in the vicinity, while the *GT2005BallLocator* at this point for the first time relocated the position of the ball into the vicinity of where it was seen for a couple of frames then with a large spread in particles quickly collapsing to the new position. So it seemed the old BallLocator did not model the movement of the ball at all, only reacting on a new position after the ball has come to a hold already. This was one major benefit of the new *Kalman BallLocator* together with the results from the *ComboBallLocator* indicating its position and velocity modeling better in about 60% of all frames even with the old 'intuitive' methods of measurement covariance matrix adaption from the *GT2004BallLocator* still used.
To be able to compare two *Kalman BallLocators* with different sets of covariance matrix adaption functions it was necessary to modify the *ComboBallLocator* a bit, now able to run the same BallLocator solution twice with an interface to have debug drawings and parameters differing in name by a prefix indicating if it belongs to *BallLocator 1* or *BallLocator 2*. The subtraction of the distance offset [section 4.3.3.1] had to be enabled in both solutions or disabled when comparing to a BallLocator without distance correction, because if the accuracy and stability is to be compared it makes no sense always getting the answer that the BallLocator with distance correction is closer to the position from the ceiling cam.
This way every new BallLocator was compared against the previously latest, keeping in mind the different game situations and aiming at results of over 60% better in position all frames and 'feeling' satisfied about the errors in value and direction of velocity. First the measuring of the velocity was that noisy that it practically was impossible to compare the errors in velocity (always getting similar results for all Kalman BallLocators). Later, after introducing additional improvement attempts [section 4.3.3.3] to get at least its direction closer to reality, it was concentrated primarily on these values, aiming at results of over $60\text{-}70\%$ better in direction and at least around 50% in absolute value of velocity to not sacrifice one goal for another.
The latest Kalman BallLocator reacted on a shot almost instantly, had a high accuracy in position, also when the ball was moving, the velocity was far less jumpy, its direction very close to its movement direction, a modeled static ball did not move significantly even when the robot's head was heavily panning, the ball in the model 'rolled' out very close to its real position when not seen and when turning around searching for the ball the odometry adaption finally worked and the modeled ball stayed at its global position relative to the robot, it could even track a ball running behind several other robots regularly obstructing vision.

Chapter 5

Opponent player modelling

5.1 Goal and Challenges

The direction and distance of opponent players is currently used in the obstacle model. However, the actual position of the players is not modelled, and as such a central information about the game's state is not taken into account.

While the positions of the players from the own team are known as long as the wireless communication is active and working, the opponent players have to be modelled using information gained from the camera. This year an effort was made to create a usable, consistent model of the opponent positions. However, a number of challenges emerged:

1. The jerseys are amongst the most irregular shapes on the field. They are also fairly small, and a single player wears several jersey parts at the same time, occluding parts of them at any given moment.

2. The team colors, dark blue and red, are particularly difficult to detect. The dark blue, in particular, is too dark for the camera to discern it from simple black in many circumstances. It's still light enough to border on the light blue in some cases, though. The red, on the other hand, is easier to see from a distance, but sometimes borders on other color types such as the pink of the landmarks. As usual, a high-quality color table is crucial to detecting robots, and in some cases a trade-off between player detection and the detection of things as crucial as the landmarks is unavoidable.

3. Visually telling one opponent player from another is impossible: While every player has a mandatory number sign attached to the body, the camera is nowhere near good enough to detect it. Effectively, all players from one team look the same.

4. All parties involved are moving: Both the observing robot as well as the opponent players are constantly on the move. In particular, this affects the field of vision, and even with perfect detection, any given opponent will only be seen for brief moments at a time. This momentary information has to be reconciled with the previously gained data.

5. The visual information is fragmentary, as no player can ever see all opponent players at one point or even within a period of time.

6. While not specific to players modelling, the very limited resources of the AIBO are always an issue. The players model had to run on CPU time gained from optimizing other subsystems, severely limiting the options for model choices.

Since the focus of this year's work was on the modelling aspect, problems 1 and 2 were taken for granted as the model relied on the detection efforts made in previous years. These problems lead to a very challenging modelling task: even under ideal conditions, red robots close to the observer were detected only 50% of the time, while red robots more than a meter away were typically not detected at all. As to be expected, detection of the blue players was still worse. When robots were detected, the detected position was not very reliable and in particular often jumped backwards on the line of sight. The orientation on the field of a detected robot is not known at all.

Problems 3 through 6 were the focus of this year's research in players modelling. The inability to tell players apart made using standard modelling approaches difficult or inconclusive, and the problem of data association had to be addressed. This is aggravated by the opponent players movement. Unfortunately, the detection and modelling quality is not high enough to associate opponent players with a speed vector, which would help in predicting the movements.

The player model is stored in absolute (ie. field) coordinates to facilitate a common coordinate system across all robots. This enable sharing the information with other players, and ideally, would enable the team to cover large parts of the field, despite the fact that the range of detection is so low. In addition to this, sharing data associated with the same physical robot should lead to a higher precision and mutual validation of percepts.

However, the conversion from relative to absolute coordinates requires a correct localization of the robot on the field, making the quality of the self locator crucial to the model.

5.2 Particle Filters

Early on, a decision was made to use a particle filter to model the opponent players. Particle filters can represent complex distributions, are fairly fast, and can be scaled in terms of their system requirements. They are also easy to implement and extend. In general terms, a particle filter represents a probabilistic description of a situation via weighted particles in the state space. Each particle is a hypothesis about one particular state, the particle's weight describes the probability of it being the correct one. The situation described by all particles taken together is referred to as the belief.

The particle filter operates in several steps:

1. Process update. In the process update an estimation of the current state is made, based on the information gained in previous states. This is usually done by moving the particles, e.g. predicting modelled position based on the position and speed modelled in the past.

2. Measurement. In the measurement update, the predicted positions are compared to the observations currently made. This is accomplished by adjusting the particle weights, resulting

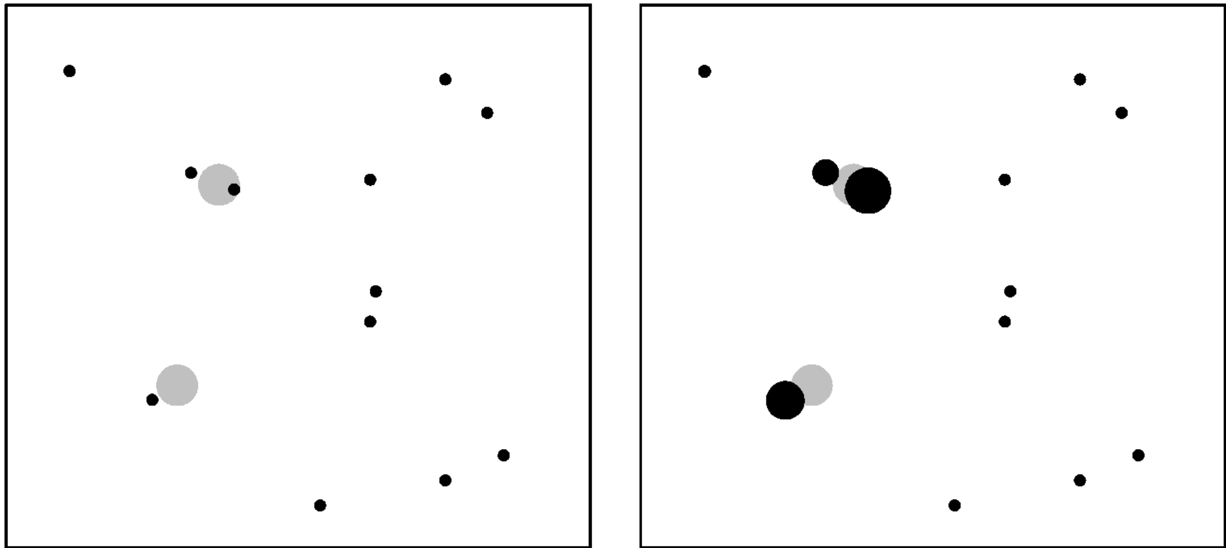


Figure 5.1: The particle field before and after the measurement update. An opponent player is observed. The weight of particles close to the percept is increased. The percept is represented as a gray circle, the particles are black with the size corresponding to the weight.

in high weights in areas where the observation model matches the particle distribution, and low weights where it does not.

3. Normalization. Since the measurement update affects the weights, it is necessary to renormalize them to ensure they sum up to one.

4. Resampling. The resampling step is not necessary at every iteration of the filter, and a simple version of the particle filter need not implement it at all. It is "a probabilistic implementation of the Darwinian idea of *survival of the fittest*"[17]: Without resampling, important points in the state space are often represented only by a single, high-weight particle, reducing the accuracy in further iterations. To prevent this from happening, high-weight particles are replaced by a number of particles corresponding to the value of the weight. Ideally, the overall belief is unchanged, but instead of representing it by single high-weight particles, it is represented by the density of particles of equal weight.

A detailed account of particle filters and their implemention is given in [17]. In particular, resampling is discussed and a fast resampling algorithm is introduced, which performs the resampling step in linear time.

The strenghts of particle filters lie in their straightforward implementation, the ability to represent complex, multi-modal beliefs and the possibility to scale their performance by adjusting the number of particles. On the other hand, for high-dimensional state spaces, a very high number of particles is necessary for even minimal accuracy. An alternative would have been to use the Kalman filter, possibly a multiple hypothesis variant.

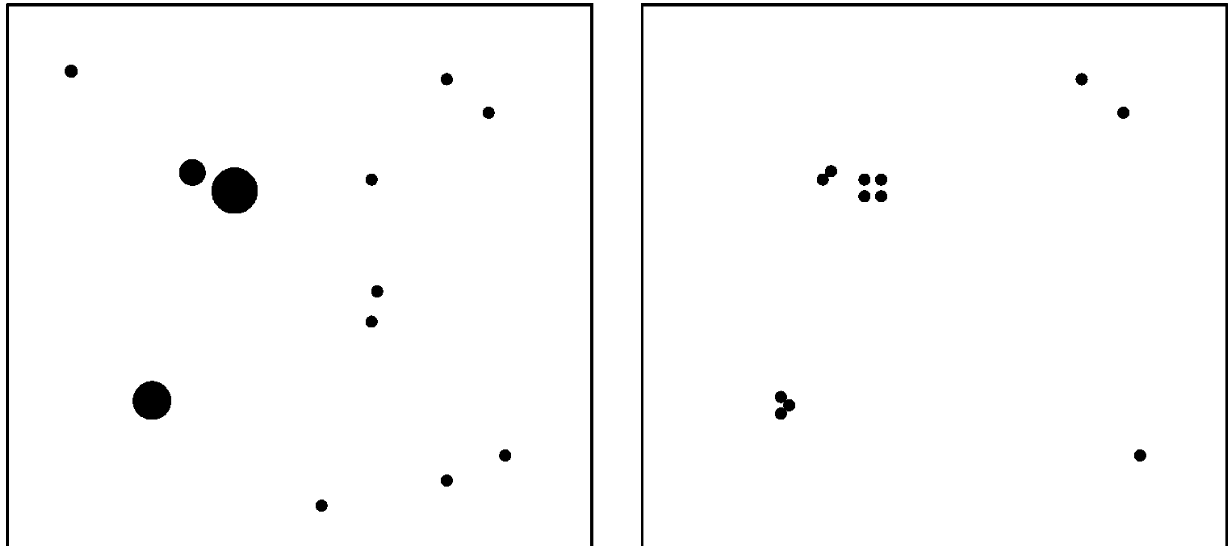


Figure 5.2: The particle field before and after resampling. High weight particles are replaced by a corresponding number of low-weight particles.

5.3 Clustering

Of course, other parts of the cognition are interested in actual robot positions instead of dozens of particles. So in order to produce useful output data, the particle field needs special post-processing. A simple k-Means clustering [7] algorithm has proven to produce decent results in a very short time. However, the number of clusters, ie. the number of modelled robots has to be produced as an input to the algorithm. An estimate of this value was arrived at by comparing the percepts to the previous results of the clustering. Other clustering algorithms do not require this additional estimate, but the hierarchical clustering implementations often had worse results and most importantly a prohibitively worse runtime than the k-Means algorithm.

5.4 Single Filter Modelling

The first approach that was extensively experimented with was trying to model all players within a single particle filter. The state space of the filter was simply the two-dimensional field. An extended version could add a third dimension for the orientation of the modelled robots, however, since the percepts do not include any reliable information in this regard, this possibility was never explored.

In the prediction step of the filter, every particle was moved randomly according to a Gaussian distribution. This is a very simplistic prediction step, but lacking any kind of odometry for opponent robots, and with the velocity information being much too unreliable, this was the only possibility.

The measurement update integrates new information into the model by increasing the weight of particles close to detected robots. The degree of the increase was based on a two-dimensional Gaussian function, identified by its center and its variance.

1. The first approach was to center the Gaussian on the percept positions. Ideally, the variance would be based on the validity of the percepts, however lacking any validity, the distance from the observing player to the percept was used instead. For a given particle and a given Gaussian based on a percept, the weight increase is computed by evaluating the Gaussian at the particle position. This is done for all particles and all percepts.

2. The second approach was a variation of the first: the Gaussians are centered on each particle instead, with a variance based on the particle weight. The increase is computed by evaluating each percept's position within this Gaussian.

As an alternative to increasing the previous weight, the computed value can also replace the weight of the particle, or an average between the two can be used. In the end, the first approach was used, while the computed Gaussian evaluations were added to the previous weight of each particle to give the new weight. This had the effect of leaving particles without validating percepts with the same weights as before, increasing those with validating percepts by depending on their distance to them.

However, increasing the weight of some particles raises the sum of all weights above 1, so in the normalization step, particles unaffected by the incorporation of percepts are reduced significantly. Any subsequent resampling step would likely remove those particles with reduced weights. Effectively, a given simulation step is dominated by the current percepts, ie. the filter only accurately models the last seen robot. These observations closely match those made in [17]:

> If, for example, one object is occluded, the samples tracking this object obtain significantly smaller importance factors than the samples tracking the other objects. As a result, the occluded object gets quickly lost, since the samples tend to focus on the other objects.

Several attempts were made to improve modelling while retaining a single filter concept. One idea was to prevent particles associated (by the clustering algorithm) with one robot from affecting particles associated with another robot, especially when normalizing. Instead of summing up over all particle weights, each group of particles is considered independently and their particle weights normalized without any interaction from other particle groups. Another approach was to disregard particle weights to a certain degree, instead relying only on the particle distribution on the field. Particle groups that are not currently perceived get low weights, but still remain in their positions, giving information on previously perceived robots. This meant avoiding resampling for as long as possible, because the resampling step removes groups of particles with low weights and redistributes them in high-weight positions, removing the wanted information from the particle collection.

In the end the single filter approach did not result in a satisfactory model of opponent robots. The various tweaks improved the accuracy of the filter, but could not hide the fact that the approach was inherently unsuited for the type of situation to be modelled. Notably, the model

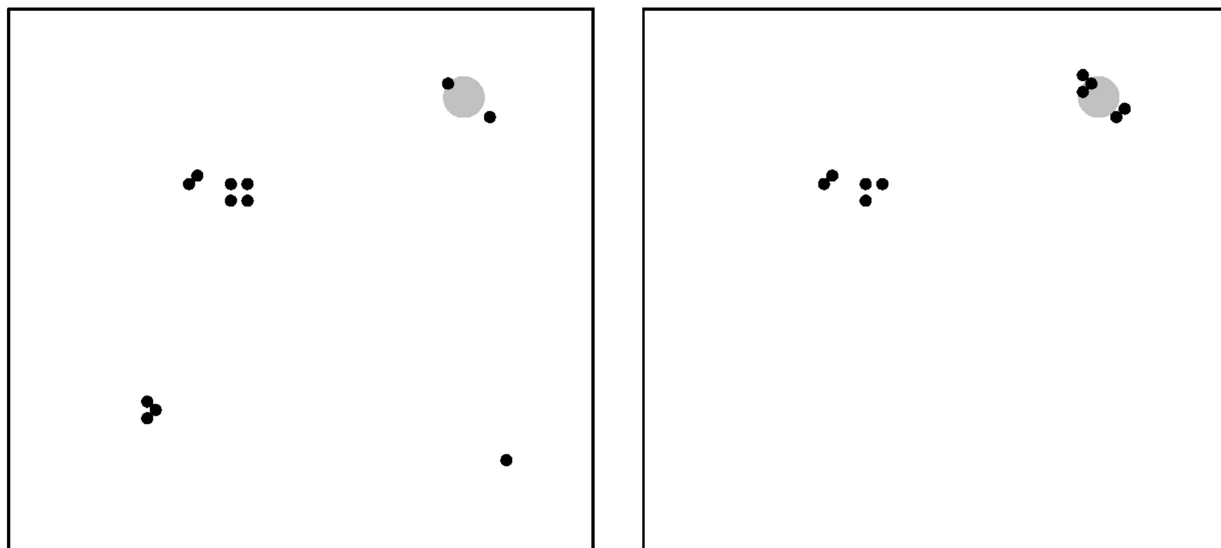


Figure 5.3: A particle field before and after a set of measurement update, normalization and resampling steps, exemplifying the information loss in the single filter approach. The measurement update increases the weight of particles close to the percept, the normalisation step reduces all weights to ensure that the weights sum up to 1. Resampling then removes the clusters in the bottom left and right corners.

worked fairly well in representing the position of a single robot on the field, it only failed with more than one player to be modelled. This is due to a wide-reaching simplification made when using only a single filter: the filter's state space was only two-dimensional, while the actual state space was of a much higher dimensionality: two dimensions for each opponent to be modelled. While in theory, a particle is meant to represent a hypothesis about the system state, in this case it represented only a hypothesis about a *part* of the system state. In a standard particle filter, the particles are mutually exclusive, that is, when a given particle accurately describes the situation, all other particles are inaccurate. In contrast, in this case any number of particles could accurately describe parts of reality, with only all those accurate particles together describing the whole of it. These relations are found again in the characteristics of normalizing, which were so unhelpful for the purposes of modelling multiple players: increasing a single particle's weight means an increased likelihood of its accuracy, and subsequently a reduced likelihood of other particle's accuracy. Thus, much of the theoretical background of the particle filter was not applicable, and the practical failure followed from that.

5.5 Multiple Filter Modelling

Clearly, forcing the high dimensional state space into a simple two-dimensional one was unviable. On the other hand, using higher dimensions brings with it other issues. Most notably, the bigger space needs a vastly higher number of particles for an accurate model, ruling out an implementation within the performance confines of the AIBO. Another issue is that of associating input data (ie. the percepts) with the model, which is now managing discrete dimensions for each

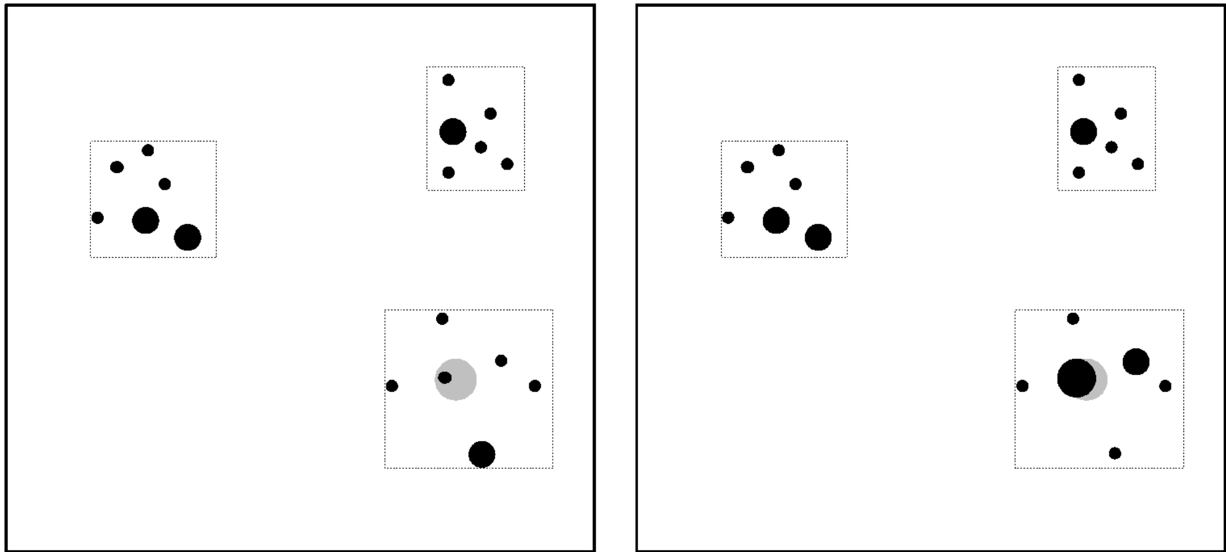


Figure 5.4: Measurement step in a multiple filter approach. The percept is only considered in the filter associated with it, the other filters are unaffected. Individual filters are represented by a dashed box.

modelled robot. Since the robots are indistinguishable from one another, the association has to be made from previously gained information, especially the position of the modelled robots: if a robot is seen close to a previously modelled one, it is most likely the same robot. (But clearly, it is also possible for this robot to have moved, with another robot taking position close by.)

An alternative to running a single high-dimensional filter is to run multiple two-dimensional filters in parallel, one for each modelled opponent.[16] The filters are dynamically generated based on an approximation of the number of robots involved. Each filter works like a normal particle filter as outlined above, but without the difficulties of having to model more than one robot position. This scales much better than an overall increase in dimensionality.

Implementing this sort of model quickly gave promising results, working much better than the single filter solution. The prediction step is very much the same as for the previous filter, simply adding random Gaussian noise to all particles in all filters.

The measurement step is different, however. A given filter should only get the percepts relevant to it as an input; since in this context each filter represents an opponent robot, it should only get percepts from this robot. However, since it is impossible to say for certain which robot is perceived, a degree of association is calculated based on the overall model state. Since the approach introduced in [16] did not perform fast enough, the degree is simply based on the average distance between the percept and the filter's particles. The input percepts are then incorporated into the existing filters, taking into account the generated degrees of association: a highly associated percept will affect the filter a great deal, a low association will have hardly any impact. One way to accomplish this is by simply using a linear interpolation between the old calculated weight and the new one, based on the degree of association. Below a certain threshold, the association can be set to zero, both to improve model accuracy and to improve performance. Apart from the effect of the association, the measurement update takes place much as described in the previous

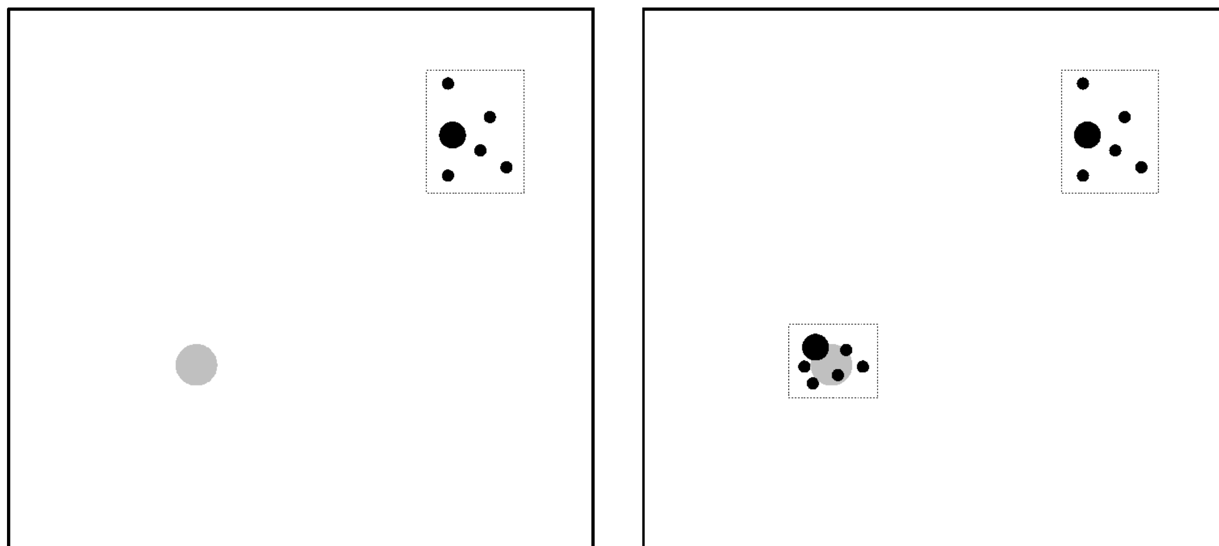


Figure 5.5: Creation of a new particle filter. The association of the percept to the single existing filter is very low due to the high distance, so a new filter is instantiated.

section: Gaussians are centered on the percepts, with the distance between robot and percept determining the variance. After incorporating the input data, normalizing and resampling takes place as per the default particle filter.

Since the number of filters varies over time, a method for creating and removing filters has to be defined. A filter is created when the sum of the degrees of association is below a certain threshold: The percept does not significantly contribute to any of the existing filters, so it is likely to represent a robot not modelled yet. A filter is deleted when it's size (in field coordinates) is larger than a certain amount. To approximate this efficiently, a bounding box is computed around the filter, and the diagonal of this box is used as the filter's size. A filter's size is increased in the process update step, and reduced in the measurement update and resampling steps, if there are percepts associated with the filter. If no percept is associated with a filter for a certain amount of time, that is, no robot is seen which is likely to correspond to the filter, the size will grow beyond the threshold and the filter is removed.

To get an estimate of the modelled positions from the particle fields of the filters, the average of the particles of each filter is computed and returned. This is much faster than performing a clustering, and along with a reduction of the number of particles for each filter resulted in an overall execution speed similar to the original filter.

5.6 Results

Under good conditions, the model provides sufficient results. Cooperative modelling, with percepts shared between robots, further improves accuracy. However, in the hectic confusion of a real game, less input data is generated, and the model tends to either err on the side of too many or too few modelled players, depending on the thresholds for the creation and removal of fil-

ters. Further work needs to be done in that area, creating as robust a model as possible under the circumstances. A better players detection would make creating that model much more easily achievable.

Specific improvements are possible. The current approach uses multiple parallel particle filters, but could use any given filter in parallel, including Kalman filters. Since the particle filters tend to represent uni-modal Gaussian distributions, the computational overhead compared with Kalman filters is not warranted: if a uni-modal Gaussian distribution is sufficient, Kalman filters should be used instead. A more accurate calculation of the degrees of association might lead to better results, but could also be more costly in terms of performance.

On the other hand, any model is only as useful as it is being made use of. Feedback from engineering behaviors can be used to improve and adapt the model to fit more specific needs, as was seen in the RoboCup Open Challenge. This is especially true for a model where due to technical limitations certain trade-offs are unavoidable.

Chapter 6

Behavior Learning

6.1 Specification of the Problem

The various different options of the BehaviorControl module require a very high level of flexibility and adaptability, because the entire demeanor of the robots depends on the progressiveness and the optimal adjustment of all those options that are used in a game.

6.1.1 Motivation for Learning on a Robot

In dynamic environments like robot soccer, it is a challenge to constantly adapt to all kinds of changes. The rules are revised regularly, and the game environment (lighting conditions and carpet surface) differs widely depending on the location. This necessitates a continual reexamination of the BehaviorControl module, for both basic behaviors and higher level skills. Accordingly, it would be preferable to be able to adapt behavior solutions as expeditiously and autonomously as possible.
With this objective, we reckoned that autonomous learning techniques could meet our needs. They offer an opportunity to save time and also to generate improved or even optimal solutions in comparison to the hand tuned code.
Our first idea was to create as universal and portable a tool as possible, in order to process all behavior options that have to be tuned in a similar fashion. After a closer look, this conception was dismissed, and it was decided to only alter some constitutive numeric parameters in the respective options by means of learning their optimal values.
Hence, we firstly chose one important option for a starting point: the "grab-ball" behavior.

6.1.2 Optimization of the Behavior "grab-ball"

The behavior "grab-ball" is used to trap the ball between the forelegs and pinch it with the head (i.e. the chin). To start by optimizing this behavior made sense because of the fundamental importance of ball grabbing skills. The only moment that the ball position is really well known is when the ball is grabbed. Additionally, opponents will not be able to get to the ball when it is

grabbed. Another big plus is the fact that parts of the ball are covered and, depending on the ball locator of the opponent teams, it cannot be detected as easily anymore.
Before the optimization, the ball grabbing was sometimes not reliable enough and very prone to fail under changed circumstances, even when the changes were only slight. In order to avail ourselves of the numerated benefits, we had to make the grab more dependable and easier to revise.

6.2 Different Learning Methods

Learning in general is a mapping from sensory inputs to control actions or, in other words, it is the modification of behavioral tendencies by experience. The goal is to acquire knowledge automatically from training, i.e. from interaction with an environment and from the observed consequences. This way, the neuromuscular junctions in the brains of living beings can be imitated, and the robots can "recall" efficient behavior patterns.
We examined different approaches that might be suitable for our optimizations, such as reinforcement learning and evolutionary strategies. The layered learning approach that the UT Austin Villa four-legged team applied successfully to learn a grasping behavior was also surveyed [6].
The use of the ceiling camera above the field in the lab for evaluation purposes was discarded, because there are no such cameras provided at competitions either, and convenient behavior revision was one of our most important concerns after all.

6.2.1 Reinforcement Learning

Reinforcement learning is a generalized approach on learning. The basic idea is to have a mechanism with rewards and/or punishments to alter numeric parameters in a controller, which leads the optimization in the right direction. The learning usually occurs in such a way that an agent acts on its environment and at the same time perceives its state. In reaction to the agent's actions, the environment provides rewards or respectively punishments that can in turn be observed by the agent again. Evidently, the intention is to maximize the reward the agent obtains.

6.2.2 Evolutionary Strategies

Evolutionary strategies are

> one of the main branches of evolutionary computation [... which] aims at benefiting from collective phenomena in adaptive populations of problem solvers underlying birth and death, variation and selection, in an iterative, respectively generational, loop. [2]

The search space is explored by applying genetic operators such as mutation, recombination, selection, and crossover to populations of controllers, leading to an optimization of the control strategies.
The function of these operators is usually to adjust the *parent* individual by adding random values

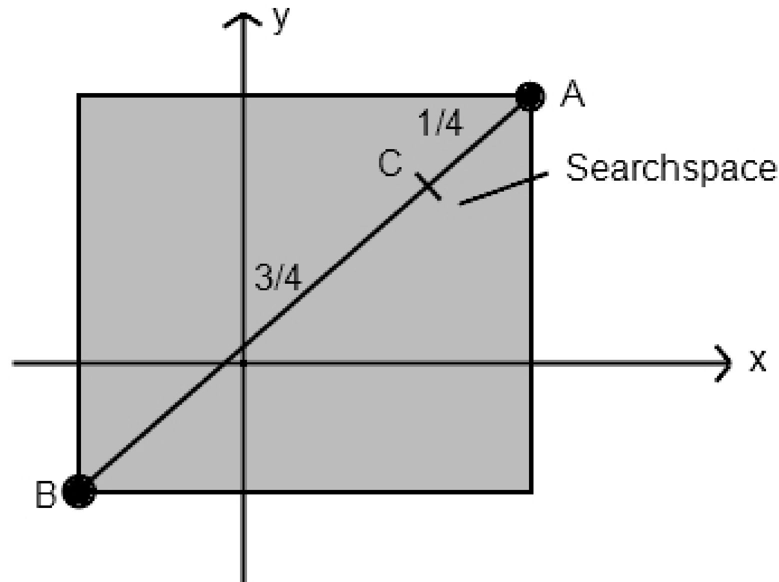


Figure 6.1: First step of an ESSE for a 2-dimensional search space; FITNESS(A)=3, FITNESS(B)=1; x and y are the dimensions of the search space.

and, depending on the fitness of the resulting *offspring* individual, to repeat this process with the respective fitter individual. Each repetition is regarded as one generation, and the cycle does not end until a termination criterion is met (i.e. an optimum is reached). Often, the genetic operators are applied to whole populations instead of single individuals, this way the strategy is less likely to proceed into local optima.

6.2.3 Experimental Search Space Exploration (ESSE)

The "Experimental Search Space Exploration" algorithm is a highly experimental and non-proven effort that tries to test the outermost points of the search space, to connect them, and to calculate new points that lay on the line, depending on the quality of the solutions. For this, the algorithm expects the best solution to be nearer to good solutions than to bad ones.
In the first step, the outermost points of the search space are tested. Then, every point is connected with the point that is on the opposite side of the search space (see figure 6.1). Dependent on the quality of the fitness function, new points are chosen. The new point C is found by starting at a point A and adding a vector $\overrightarrow{AB}$ with the length

$$l = \frac{FITNESS(B)}{FITNESS(A) + FITNESS(B)} \tag{6.1}$$

$FITNESS(A)$ and $FITNESS(B)$ are the fitness functions at the points A (or B).

In the next step, the new points are tested and connected in order to get new points again. After testing and connecting these new points, the very last single point to test is obtained. In our

tests, the best obtained point was always a good approximation for the best value (found with the help of the other optimization method).
The advantage of this algorithm is its well defined execution time. Only $2^{n+1} - 1$ (n = dimension of the search space) different values have to be tested in order to get a good approximation. This makes it useful even at tournaments.

6.3 Description of the Chosen Approach

The following sections describe the way the optimizations and learning methods were implemented for effective operation on the robots.

6.3.1 Behavior Evolution with Reinforcement Learning

For our application, a combination of evolutionary methods with reinforcement techniques showed to be most convenient, since human interaction and supervision unfortunately seemed unavoidable. The optimal behavior, specifically the target state of the grabbing action, was known a priori, but the robot had no explicit notion of correctness that could help it to evaluate reliably whether it had achieved this target state. After all, the robot could not see the ball once it was grabbed underneath its head. The infrared chest sensor could be used, of course, to check whether the ball had been captured successfully. But this method was not completely faultless, and the accuracy of the ball's position under the chin could not be judged by it.
That was why the supervision of the learning process by humans was so important. Thereby, the attempts that were non-promising could be discarded right away, without having an impact on the fitness evaluation itself.
So we decided to use an evolutionary strategy to alter the parameters of the behavior, and we integrated reinforcement principles to speed up the optimization and to allow human interference. Actually, we implemented two different optimization strategies within the learning behavior, one associating a very simple evolution strategy, and one according to the depicted ESSE principle. The algorithm that is applied can be switched easily.

6.3.2 Parameterization of the Behavior "grab-ball"

Our objective was to maximize both velocity and robustness of the behavior while avoiding to lose the ball. So we had to make out those features that were relevant to this learning task.
For the grabbing procedure, the robot runs to the ball and then stops at a specific distance and puts the head down. If it is too close to the ball, it will kick it away. But if it is too far away from the ball, the robot touches the ball only with its chin and, by this, pushes it away. Also, if the robot does not approach the ball facing it within a certain angle, it will likely kick the ball away with the paws or the forelegs before it can reach the appropriate distance for the grab. Yet another problem is the time that is needed to grab the ball. Every second the ball lays uncovered on the field, it can be seized by the opponent robots. So it is important to reach the ball and to execute the grabbing action as quickly as possible.

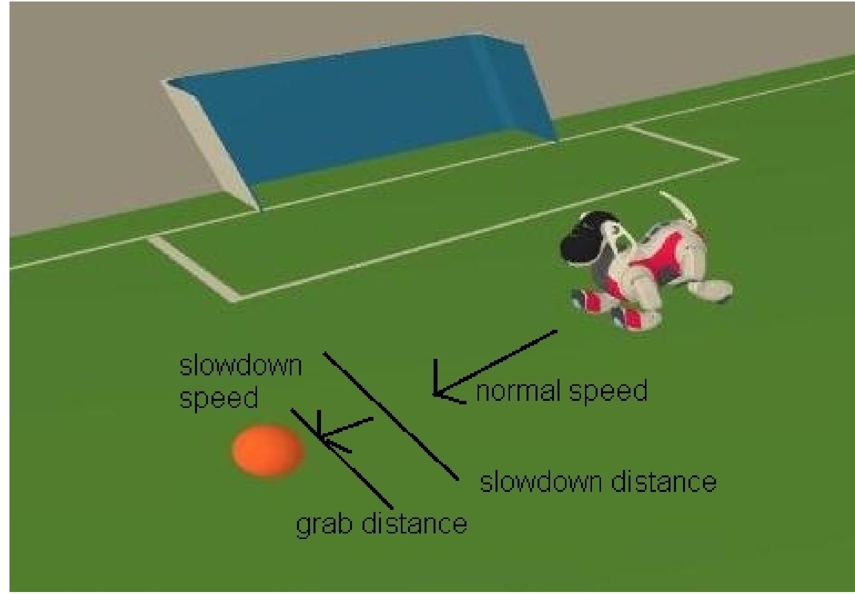


Figure 6.2: Different speeds and distances of the behavior “grab ball”

This leads us to the parameters that have to be optimized: the first one is the distance to the ball, at which the grab should be initiated (**grab_distance**); the second parameter is the proper angle that the robot has to maintain when going to the ball (*grab_angle*); we decided we could disregard this *grab_angle*, because it was already taken care of by the ball-approaching behavior. The last two parameters are related to the speed that the ball is approached with. To make the grab more accurate, this approaching process is split up into two phases. The first phase is the general approaching, where it is most important to get to the ball fast. The second phase is the slowdown-phase, where the speed of the robot is reduced in order to increase the accuracy. The two parameters in these phases are, on the one hand, the distance at which the robot begins to slow down (*slowdown_distance*), and on the other hand, the factor by which the approach is slowed down at the specified distance close to the ball (*slowdown_speed*). Figure 6.2 illustrates these parameters.

6.3.2.1 Calculation of the Fitness Function

The fitness function accounts for the quality of the chosen parameters and consists of two components. One is the time that the robot needs to grab the ball. Because of the importance of being faster than other robots, this value has to be minimized. The second component is the quantity of successful grabs. This value should to be maximized, obviously.
Since we decided to do a minimization, our formula for the fitness was:

$$FITNESS(x) = \frac{time(x)}{successfulGrabs(x)}, \forall successfulGrabs(x) \neq 0 \tag{6.2}$$

Figure 6.3: The training procedure for the behavior “grab ball”

6.3.2.2 Training Procedure

To train robots in ball grabbing, we positioned two balls on marked points on the field in a distance of about two meters from each other (see figure 6.3). The robot started in the middle between the balls, facing either one of them. Then, it had to walk up to the ball it was facing and made an attempt to grab that ball. After the attempt, the robot turned around and started over with the second ball, while the first ball was taken back to its marked position in case it had moved during the attempt. This procedure had to be repeated alternately with the two balls until the robot had counted thirty tries, then new value sets were calculated for the next training set, according to the applied optimization strategy (see below).
At competitions, we reduced the count of tries to ten. This is still enough to give an impression of the quality of the current parameter set, but it speeds up the training significantly.

6.3.3 Optimization strategies

Two different algorithms were implemented for the actual alteration and optimization of the parameter values:

6.3.3.1 (1+1)-Evolution Algorithm with 1/5th-Rule for Mutation Strength

A (1+1)-evolution algorithm is a simple evolutionary strategy which only takes one parent and one offspring individual into account in each generation. Mutation is performed by adding Gaus-

sian random variables simultaneously to each parameter. These variables are, in our case, from a normal distribution with a mean of zero and a standard deviation of sigma (sigma was initialized with five).
Additionally, we deployed the 1/5th-rule for adaptation:

> *In order to obtain nearly optimal (local) performance of the (1+1)-ES in real-valued search spaces, tune the mutation strength in such a way that the (measured) success rate is about 1/5.* [2]

To regulate the mutation strength in the described way, the standard deviation sigma was multiplicatively adjusted depending on the ratio of successful mutations to the overall number of generations. As factor for this adjustment, we used 0.85 as recommended by Hans-Paul Schwefel in [2]. Whenever the ratio was larger than 1/5, sigma was divided by this factor; when the ratio was smaller than 1/5, sigma was multiplied with the factor; in case the ratio was equal to 1/5, sigma remained unchanged.
As condition for the termination of the learning process, we counted those generations, where the parent individual had a better fitness than its offspring. When this happened consecutively five times in a row, we presumed to have found an optimum.

6.3.3.2 Experimental Search Space Exploration (ESSE)

Since our search space was a three-dimensional one, we had to do fifteen runs of the ESSE. For the slowdown-distance, the values range from 100 to 150 mm. For the slowdown-speed, the borders are 10 and 400 mm/s. And the grab-distance lays between 100 and 150 mm. With these extreme points, the first eight runs were started.

6.3.4 Button Interface for Manual Award and Mutation Direction

Additionally, we created a button interface that was used to punish the robot manually when its own evaluations failed, and also to influence the direction of mutation of the parameters. This way, the training process sped up significantly, but it still took several hours to produce feasible results.
We added sounds to the learning behavior in order to be able to recognize whether the robot deemed its current attempt successful or not. The detection of failed attempts was done very reliably by the robot itself, but it would also judge attempts to be successful where the ball was grabbed only laterally or too much up front. In these cases, we had the possibility to impute a punishment to that attempt by touching either one of the three buttons on the robot's back right afterwards.
The most significant one of the optimization parameters was clearly the distance to the ball when it was grabbed. By observing the robots while they were attending their training sets, it was unexacting to apprehend whether the current value of the *grab_distance* brought them too close to the ball or let them stay too far off it. We could detect if the robots kept pushing the ball away with their chest or penning it with the chin too much up front. In these cases, we

intervened into the next modification of the parameter by setting the mutation direction manually with the button interface. After a training set of ten or thirty tries, the robot would wait for a couple of seconds to receive a mutation direction. If we pressed the hind back button during these seconds, the mutation for the next *grab_distance* added a positive random number to the parent parameter; if we pressed the fore back button, a negative random number was added to the parent *grab_distance*. If no button was pressed while the robot was waiting, the mutation proceeded in the original manner.

6.3.5 Other Improvements

Write values to file A complete optimization takes several hours. But the battery can supply the robot only for about twenty minutes. So it is important to write the computed values to the memory stick. Therefore, a new file is created, and the parameters and fitness values are written to this file. When the battery is changed, these values can be read out again and the optimization can continue.

Head control mode We found out that the head control mode used for grabbing was not optimal. There was always a small gap between the chin of the robot and the ball, and the ball control was precarious. Also, the mouth of the robot was closed. So we opened the mouth and lowered the head a bit. With these means, we achieved that the ball was literally pressed in between the robot and the floor and guided safely when the robot moved. Thus, we were able to further improve the grabbing and made it more reliable.

6.4 Evaluation of the Results

By the learning methods that we implemented on the robots, the behaviors for ball-acquisition could be improved significantly. More importantly, we found an adequate way to make those behaviors easily adaptable to changed circumstances.

6.4.1 Concrete Analysis

After our optimizations, we assessed that almost ninety-five percent of all grabs are successful now. Of course, this very high success rate is not always observed in games, because there the robots are seldom unhindered when they are trying to capture the ball. And changes in other modules of the code can impact the accuracy of the grab as well. Hence, the exact improvement that was reached by the learning techniques cannot be accounted for precisely.
It was remarkable to see that two of the optimization parameters which we had chosen did not seem to affect the ball grabbing accuracy severely, unlike we had assumed. The robot did not have to slow down in order to be able to grab the ball, and so the values for *slowdown_distance* and slowdown_factor that were learned differed widely in our experiments. Similar observations had been made by Peggy Fidelman and Peter Stone before (confer [6]).
Eventually, the *grab_distance* was not only the most significant one of our chosen parameters,

but also the only one that really drew distinctions in the learning process.

6.4.2 Comparison of the Implemented Optimization Strategies

Both strategies led to nearly the same values; the main difference between them was the execution time.
The execution time of the ESSE-algorithm is well defined: it takes exactly $2^{n+1} - 1$ tries (n is the number of dimensions in the search space). In all of our experiments, the distance of the values that resulted from the ESSE to the optimal values obtained by the (1+1)-algorithm was very small. But in contrast to the ESSE-algorithm, the (1+1)-algorithm does not have a well-defined execution time. In our test, the (1+1)-algorithm found good results only after a long execution time. So the ESSE-algorithm was faster and led us close enough to optimal values.
But the disadvantage of the ESSE-algorithm is that it is not proven to find the optimal results. Only our experiments show that the algorithm seems to work correctly. However, the (1+1)-algorithm is well researched.
This led us to the idea to get a first approximation of the values by using the ESSE-algorithm and to use the (1+1)-algorithm afterwards with the computed values as initialization. Thereby, we could reduce the optimization time and nevertheless get the optimal values.
For competitions, we predetermined the values in the lab and then only used the (1+1)-algorithm for the last adjustments on the competition carpet. This way, we found the respective values in a few hours.

6.4.3 Conclusion

The evolution of behaviors proved to be very useful for training complex skills that are difficult to fine-tune manually. It was even discovered that some parameters (like *slowdown_distance* and *slowdown_speed*), which were originally believed to be essential, actually had no appreciable effect.
Based on this, a couple of needful options that imply ball grabbing could be tuned very efficiently at the RoboLudens in Eindhoven in April and at the Robocup world championship in Bremen in June.
Finally, these prospects encourage to continue the work on learning methods for behavioral tasks and to identify more areas where they could be utilized.

Chapter 7

WalkingEngine

In order to play robot soccer efficiently the robots have to be able to manoeuvre fast and precisely. Playing in the 4-Legged-League the robots have legs instead of wheels for their locomotion. The gait of the movement can be controlled by controlling the joints of the legs. The task of the WalkingEngine is, to generate a set of joint parameters in each timeframe of the motion process for a certain requested walk. As we want to have an omnidirectional movement for the robots, each requested walk consists of an x-, y- and a $rotational$ component. In the recent years of development in the 4-Legged-League, the WalkingEngine has been improved several times. Today we have a fast and stable working WalkingEngine which makes the movements of our robots competitive for playing robot soccer.

7.1 Description of the existing WalkingEngine

In order to generate a walk, the most efficient and straight forward approach is to use the model of the wheeled locomotion. Thus every leg has to be moved on a path that is similar to the movement of a wheel. In the phase, where the leg touches the ground (the so-called "ground-phase") the robots COG (Centre of Gravity) moved to the opposite direction of the leg's movement. The fastest gait for walking with four legs is to move two legs, which are positioned diagonal to each other, at the same time. The other pair of legs is moved shifted in time, in the second half of the cyclic gait. This way it is guaranteed, that two legs always touch the ground and the movement proved to be stable enough for the *AIBO*.

The path of the movement of the legs is described by a polygon with theoretically an arbitrary number of vertices in the tree dimensional space. Considering the precision of the robot's joint components and computational power, it emerged, that the use of polygons with more than four vertices would take too much processing time. Therefore different polygons with four vertices are used. The outer appearance and the weight distribution of the robot is almost perfectly symmetrical to the x-axis. Because of that, the same polygons are used mirrored for the corresponding front and hind legs.

Additionally to the position of the vertices in space, the share of time for the movement along the corresponding edge in percentage of one complete round is stored. Of course the time

for one complete round, the gaits frequency has to be stored on the robot either, this is called the steplength. The polygons are optimised with methods discussed later in chapter 7.6 . A polygonset for straight forward movement with the highest possible speed is stored on the robot.

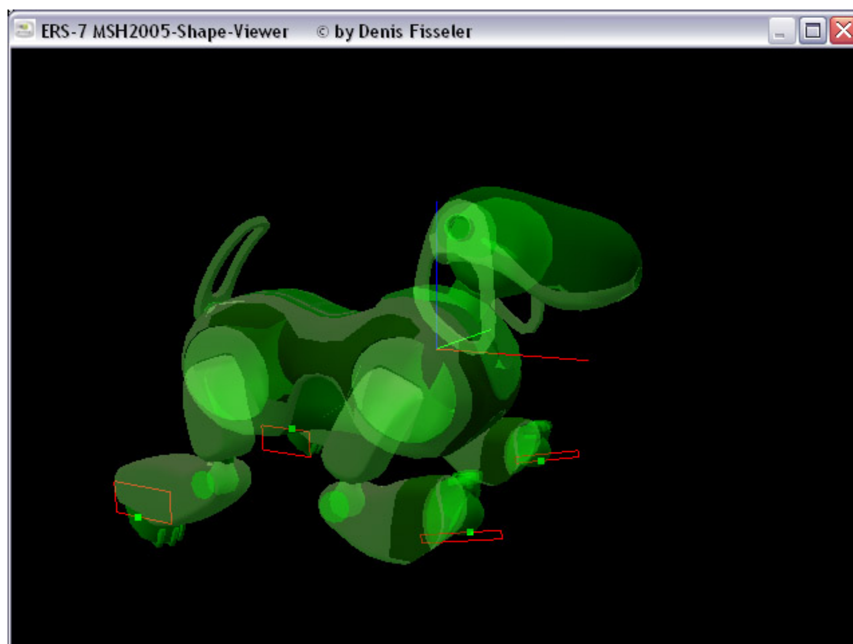


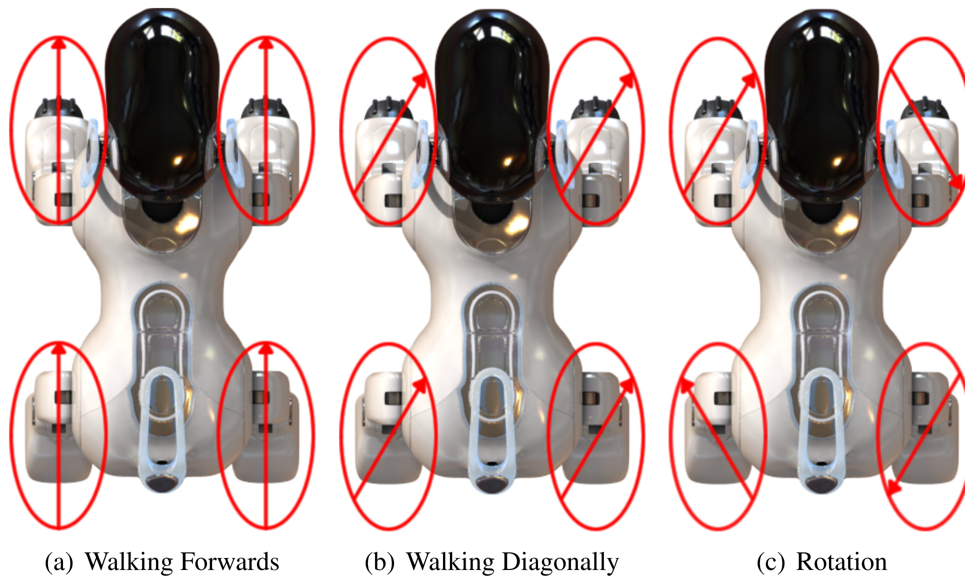
Figure 7.1: WalkPolygons

In order to change the speed of the walk the polygons have to be stretched (or in this case compressed) due to the ratio of the change in speed. To change the direction of the walk, in order to be able to walk sidewards or diagonally without changing the robot's orientation on the field, each polygon has to be rotated about the positive z-axis of the centre of the polygon. A pure rotation can be reached by rotating the polygons about the z-axis and changing the direction of the polygons, as shown in 7.2

For different types of walks, different polygonsets are used. Even a single walktype is composed of diverse polygonsets. The polygonsets have been optimized separately, which is discussed in chapter 7.6 . Because the method used for of the robot's walk is much too complex, the resulting speeds can not be predicted from the data of the polygonsets. This way the odometry has to be measured experimentally.

7.2 Problems of the existing solution

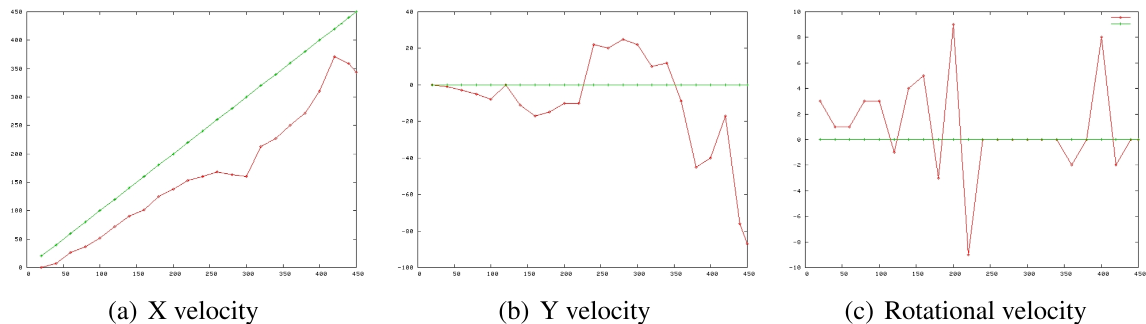
Observing the walks of different robots, there seemed to be a significant deviation between the controlled values and the real movement. This was especially striking for walks with no rotational component. Executing these requests, the robots clearly drifted into one direction and their orientation was changing on the field. There were also significant differences between the robots. Especially the older robots differed a lot in their walking behavior. It seemed to be necessary to



(a) Walking Forwards (b) Walking Diagonally (c) Rotation

Figure 7.2: The Wheel Model for different WalkRequests

measure the errors in the WalkingEngine. The CeilingCam was used to do these measurements as we assured, that the accuracy of it would be sufficient enough to record data of the robot's movement.



(a) X velocity (b) Y velocity (c) Rotational velocity

Figure 7.3: Sample measurements of the odometry

The measurements in 7.3 show the taken data for straight forward *WalkReqests*. The figures present the average measured velocities for each x, y and rot velocities for walks with x component only. The deviation form the targeted speed can clearly be observed in 7.3(a). Observing the results for different *WalkReqests* led us to the conclusion, that there is a systematic error for each velocity which can be modeled as an additional constant rotational offset.

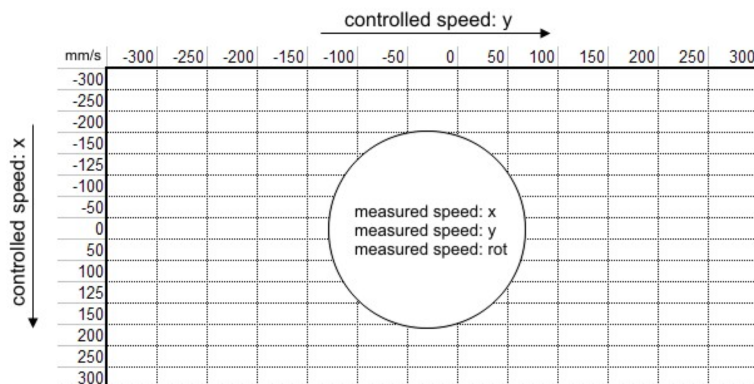


Figure 7.4: Schematics of the odometry table

7.3 Approaches for the correction of the WalkingEngine

The implemented solution for the WalkingEngine is based on premeasured odometry values for the *AIBO* robots. This is why a working WalkingEngine is as good as the quality of the measured odometry. In order to get a well-working odometry table, the main question to answer is: What is wrong with the existing solution? The odometry table, measured last year, consists of seven different tables each containing data for a specified controlled rotation. In detail, one table offers information about the controlled and the measured speed of the *AIBO*, concerning $rotation$-, x- and y-velocity. As shown in figure 7.4, measurements for the omnidirectional walk were taken for 13 predefined speeds for the y direction and 15 speeds for the x direction with a constant controlled rotation. This results in a data pool of 585 controlled and 585 measured speeds per table. Measuring the data for seven new tables again, to gain a new and precise odometry table, was no solution of the problem. As mentioned above, the *AIBO*s differ a lot in their walking behavior and the odometry table depends on data, which does not correct the differences between all robots. Creating different odometry tables for each robot, was no solution either, because the *AIBO* robots are aging very fast and the possible precision of the odometry data is within limited barriers. In addition to it, the measurements for 1365 different *WalkReqest*s had a systematic error. During walking, the power consumption of the *AIBO* is very high. This has a non neglecting influence on the walking behavior over time. Because, if there is low power, the robot moves more slowly and has a tendency to walk in a great curve, instead of walking straight ahead. In respect of this information, it was held clear, that creating a perfect odometry table was quite impossible. A most common solution was the improvement of the existing WalkingEngine. Several approaches have been considered. The first approach was an extension of the WalkingEngine, which corrects the individual walking behavior for each robot. To do this, data of the walk characteristics have been measured and stored in an additional table. This procedure allowed the correction of especially often used walking speeds. As there is a lot of information stored in the odometry table for walk requests, some of which result in the same measured speed. The measurements for this correction method only need to cover a smaller spectrum of speeds. Although

the chosen correction for the omnidirectional walk has never been used in competitions, it gives an idea of the complexity of the existing problem.
Another solution for gait optimization deals with odometry table inversion. Because the robot's odometry information is a non predictable result of the current *WalkReqest*, the new idea was, to retrieve a *WalkReqest* as a result of a specified walking speed.
The last solution for correction describes the optimization of polygonsets and the adaptation of the odometry to a new environment. New polygonsets have been developed for a walk type called *WalkWithBall*. In the following, these different solutions are described.

7.4 Correction by measurement

The polygons of the WalkingEngine are optimized by an evolutionary strategy, taking information from the CeilingCam as a feedback for the measurements. Using the CeilingCam seemed to be an appropriate tool for the correction of the walks in the WalkingEngine. The method was supposed to be kept flexible enough to be able to correct special walks that were used more frequently. The correction had to deal with the deviance between the different robots. This way, the correction could not be hard coded and had to be easily changeable for each robot. The basic idea was to continuously measure the odometry of a specified walk. This measured odometry is compared with the target walk and the error is corrected by a feedback control strategy. This is done until the measured walk is below the specified error thresholds. Whenever a result is found to be close enough to a specified target request, it is stored. Afterwards using these results in the WalkingEngine, the incoming walkrequests are mapped to the new walkrequests, which are executed. These walkrequests are supposably closer to the original requests. For the correction, a specified behavior and a special tool in *RobotControl* was introduced.

First of all, the developer has to specify a set of walkrequests. There is a set of standard walkrequests which cover the *X*-*Y* space without rotation in a manageable number of discrete steps and a set of walkrequests with rotation only, that cover all the speeds of the rotational space. If special cases are needed, which are e. g. used more often or cases where a reliable exact e. g. straight walk with a certain speed is required, those can be added here. Because the list is processed sequentially, it can be modified at any time.

The behavior is controlled from the *RobotControl* tool. It communicates the data of the CeilingCam via WLAN to *RobotControl*. In order to measure the speed accurately, the time, the robot needs for acceleration to the specified velocity, is ignored for the measurement. After measuring the speed, the deviation from the requested speed is calculated and a new request is sent to the robot. This feedback control loop is executed until the error falls below the specified error threshold. Differences in the walks were not only observed between different robots. The quality of the carpet was not the same on different areas of the field. Therefore meeting the criteria had to be verified on different sides of the field, before accepting a certain correction for a *WalkReqest*.

The acquired results are specific for the robot. Therefore they are stored in a robot special data file. The WalkingEngine is using these data to map the incoming *WalkReqest* s to new *WalkReqest* s that are matching the target *WalkReqest* closer.

7.4.1 The WalkCalibration tool

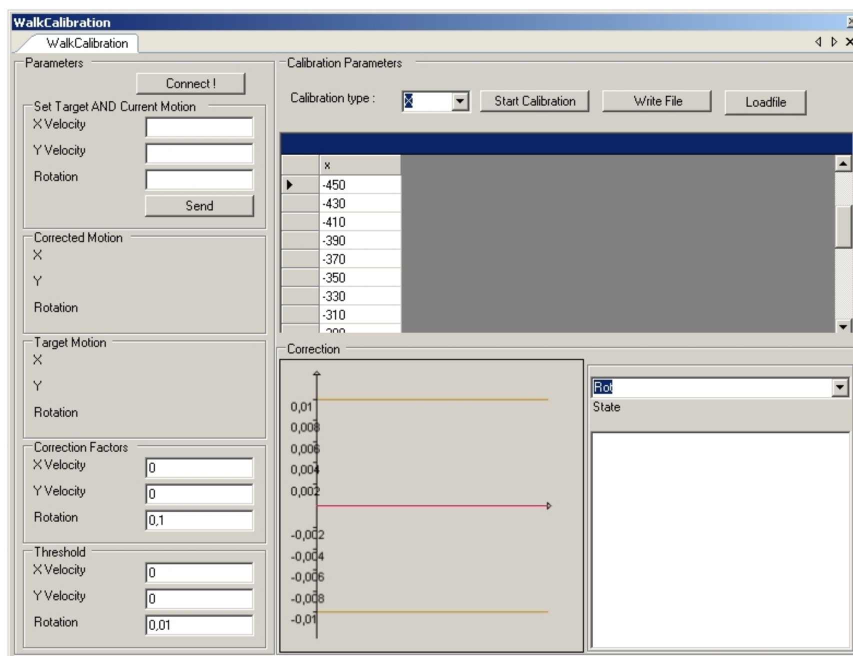


Figure 7.5: *RobotControl*: WalkCalibration

In the following, the use of the tool, which is designed for this kind of odometry correction is explained.
The *AIBO* has to use the appropriate behavior which can be controlled by the walk correction tool.

It has to be made sure, that a connection to the robot has been established, and either the behavior ‘WalkCalibration2006’ has been chosen in Makestick before the program was loaded onto the robot or *RobotControl*’s ‘SolutionSelector’ tool can be used in order to select the right behavior online. Before the measurement, the *AIBO*’s legs have always to be cleaned as properly as they will be cleaned in a competition. A dirty leg can lead to false results.

- Connect - Triggers the sending and receiving of the messages and commands from the behavior. This command can be repeated if the robot is not responding due to network problems. You receive a feedback in the output window.
- Load file - Loads data from a written file. This can be used to manipulate the set of data stored or to continue an interrupted correction
- Write file - The corrected odometry will be stored in a file specified by the user in a popup savefile dialog window. This file has to be located on the robot in the ‘mshwep’ directory.
- Start - Starts the behavior on the robot. Due to network problems, this command can be repeated if the robot is not changing its state in the behavior.

- Threshold - The error thresholds are stored for each direction. This is the criteria for accepting the specified *WalkReqest*.
- Calibration type - The type of the specified walks is selected here, i. e. for an x-Calibration, only the x and the Rotational component are adjusted. For a y-Calibration, the y and the Rotational components are changed. And finally for a rot-Calibration, merely the Rotational component is changed.
- Set Target AND Current Motion - If the tool is used to test the odometry a special walkrequest can be set here overwriting the current target walk taken from the list.

Because it is used frequently while playing soccer and only one dimension has to be corrected, the calibration of pure rotational *WalkReqest*s can be easily corrected manually.

7.5 Correction by odometry table inversion

The odometry table is mapping between the requested walk speed and the speed of the robots movement, executing that requested walk in reality. The adjustment of the real speed to the controlled speed and the correction of the walk's direction could therefore simply rely on the odometry table. The table contains a limited number of entries. In order to calculate the values in between, these values have to be interpolated. This function contains ambiguities partly caused by errors in the measurement of the odometry and by ambiguities in the walking engine. Therefore there are several different possibilities to implement a method that inverts this mapping. Trying a heuristic method by searching for the corresponding input walkrequest with a recursive algorithm, an easy and fast way was found to find the right parameters, that at least theoretically would result in the right odometry. Testing this with the existing WalkingEngine no significant changes were detected in the execution of the walk.

7.6 Correction by optimization of polygonsets

This year the development of new polygonsets for walking was less intensive than last year. Although a lot of effort was made in creating faster and stable gaits for plane walking, the search for new optimal polygons was without useful results. The polygons for normal walking, which were developed last year by evolutionary algorithms, seem to be almost optimal with some exceptions. All efforts in developing new polygonsets resulted either in equivalent or too unstable gaits. This has made clear, that a physical barrier for triggering the joints of the *AIBO* was reached.
Although the Microsoft Hellhounds had the fastest walk, including a boost walk on competitions last year, this year a lot of teams have developed equally fast walks. Having in mind, that fast and precise walking is one of the most fundamental criterion for winning a robot soccer match, a more specialized but often used gait has been optimized. This gait is used, when the *AIBO* grabs the ball and walks into a certain position or turns into a certain direction on the field. The *AIBO* should be able to move omnidirectionally, even backwards with the ball grabbed. This was

a critical point in development, because a well balanced control of the ball with the *AIBO*'s head needed to be found. The walk had to work on different carpets and enable the *AIBO* to perceive the environment. To optimize speed and ballhandling, an automated solution for development was not suitable. The robot did not detect accurate enough, when it had lost the ball and could not recover it by itself. Because of this, a manual optimization of the polygonsets was used.

7.6.1 Manual Optimization

The manual optimization of polygonsets is difficult and depends much on the intuition of the user. Although some tools, which are described later exist, the resulting movement for a changed polygonset can not be previewed. Polygonsets must be tested on the *AIBO*, to get an idea of the resulting walking behavior of the robot. In addition, the space of exploration for a three dimensional polygonset is much too large. In order not to stuck in a non deterministic search for a needle in a haystack, the search had begun with polygonsets, which were already known for the normal plane walk. This was supposed to facilitate the optimization, especially for the new developed walk with the ball. The already existing polygons have been developed by use of evolutionary algorithms and therefore ensured a sufficiently high speed. As mentioned above, a reliable ballhandling was requested for the *AIBO*'s movement. Because of that, the polygons for the frontleg movement had to be changed. When the ball is grabbed with the head robot's head, the front paws' movement and position is the most important factor for not loosing the ball. If the frontlegs are positioned too far backwards, the ball can easily slip out in the gap between the head and the legs. To stabilize ballhandling, the legs should serve as a barrier for the ball to both sides of the robot's head. On the other hand, if the legs are positioned too far in the front, touching the ball once with a paw is enough, to give it a kick and loose it. In addition, the polygonset for the hind legs were changed as well. The z-component for the axis of leg movement has been adapted, resulting in a lifting up the back part of the robot. This makes the *AIBO* lean over the ball with the head and makes sure, that the ball is not lost, when the robot moves backwards. Speed results for the new developed walk can be found in table 7.1.

Direction	**Speed**
Forward	43,64 cm/s
Backward	28,66 cm/s
Left	30,72 cm/s
Right	32,27 cm/s

Table 7.1: Results of the optimized walktype *WalkWithBall*

7.6.2 WalkshapeViewer

In figure 7.6 the interface of the *RobotControl* tool WalkshapeViewer is shown. It is used to view and optimize parameters of the three dimensional polygons for the gait of the *AIBO*. The poly-

gons consist of a set of parameters: x-, y- and z-coordinates for each vertex of the polygon and the length of each edge. Thus a polygonset can be described by 24 variables, which are displayed in the editboxes, as shown in figure 7.6. The walk polygons for the front- and backshape may differ. They can be edited separately by pressing the adequate radio button. To optimize a certain polygon, it is very useful to change the values of all parameters for a desired axis. This can be done by using the sliders for each axis. The amount of change can be selected as well. To adjust the frequency of legs' movement, the time for the steplength can be determined in a special edit box. Changing these values significantly changes the *AIBO*'s walking behavior. If the selected steplength is too long on the one hand, the frequency is low and the robot moves too slow. On the other hand, if the frequency is too high, by selecting a shorter steplength, the robot may move in a shaky way or does not move at all.

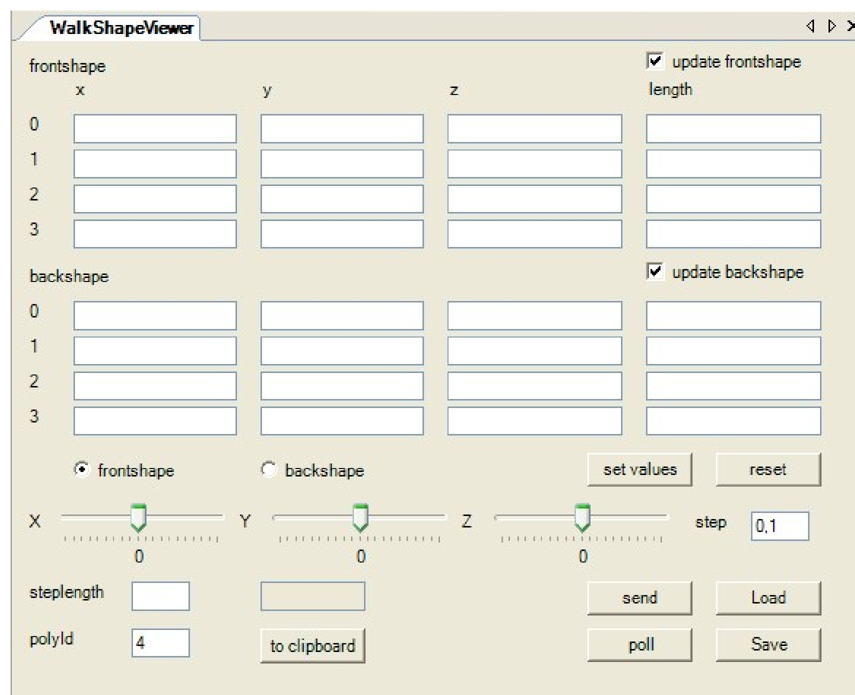


Figure 7.6: *RobotControl*: *WalkShapeViewer*

For testing, the newly created polygons can be sent to the *AIBO*, by clicking the send button. It is of great importance, that the polygon id is set to the correct value. The polygon id describes, in which order the WalkingEngine loads different polygonsets from the stick. The correct polygon id can be found, by looking into the "GT2005WalkigEngine.cpp". By selecting a wrong id, the polygonset of a different walk will be overwritten. The *WalkshapeViewer* also offers the possibility to receive different polygonsets from a connected *AIBO* by pressing the poll button. This can be useful to regain lost or overwritten values. Another functionality of the *WalkshapeViewer* is to copy a tested polygonset to clipboard. This polygonset can be used as a starting parameter for an evolutionary optimization of the gait. The copy to clipboard function generates predefined sourcecode including the selected starting parameters for existing evolutionary algorithms.

7.6.3 Evolutionary Optimization

The fastest and best method to find optimized polygonsets for walking, is to make use of evolutionary algorithms. Evolutionary optimization was both used for searching faster gaits and plane walks and for the development of some polygonsets for the new walktype *WalkWithBall*. As there already were working evolutionary algorithms, which had to be adopted to new circumstances, this chapter refers to the implementation developed last year [3].
To evolve certain polygonsets with the *AIBO*, a special behavior is necessary, to let the robot perform plane walks within the range of the CeilingCam automatically. The CeilingCam is used for measurement of the odometry and sends the information to the host computer, which communicates with the *AIBO* via WLAN. To minimize errors in measurement, the acceleration phase of the *AIBO* should not be considered. Because of that, a time period of two seconds was inserted before the measurements starts, to ensure the robot reached its maximum speed. The measurement itself is a linear point to point calculation. The behavior for the *WalkWithBall* evolution has been changed and differs slightly from that, which was used last year. It is created for haptic feedback of the supervisor and now is semi automatic. This has been done, because during evolution the robot has lost the ball very often and had to be placed on the field manually after each walk.
When a new fast polygonset has been evolved, the measurement for this specific polygonset is repeated a second time. As the information on the robot's position from the CeilingCam possibly had some errors (was noisy), this way the error is minimized.
Evolutionary algorithms have been used to develop sidewalk polygonsets for the new walktype *WalkWithBall*. Although manual optimized polygonsets have been developed, which were faster than those developed last year, optimization by evolution maximized the speed again. Results are listed in table 7.1.

7.6.4 Customize Odometry

Figure 7.7: *RobotControl*: *CustomizeOdometry*

Having optimized polygonsets for fast and precise walking on the one hand, reliable odometry information on the other hand is a prerequisite for a well working WalkingEngine. The odometry data is used by the *AIBO* to calculate its position for selflocalization. Different carpets with different quality cause a variation of speed and a wrong calculation of the robot's position on the field. Measurements taken from the field in Dortmund and Bremen have shown, that there exist differences in speed up to 13%. A reason for this is the friction of the *AIBO*'s legs on the carpet. Because it is not possible to develop a WalkingEngine with correct odometry information for any terrain, the *AIBO* robots need to be calibrated on different carpets for competitions. There is also a difference in speed between the older, used robots and the new ones. This must be taken into consideration as well, when measuring and calibrating the odometry.
This part describes the calibration for the new developed walk *WalkWithBall* with regard to the *RobotControl* tool *CustomizeOdometry* and the calibration behavior *Customize-odometry*. Due to experiences on a lot of competitions, the WLAN for testing and adaptation purposes does not work well most of the time. Therefore the main condition for the development was, that the calibration should be independent from a working WLAN connection to an *AIBO*. On competitions, there is also no possibility to make use of a CeilingCam. This makes the measurement and correction for all omnidirectional walking speeds impossible. Because of that, the odometry correction is reduced to the most frequently requested speeds. In this case, for the new walk *WalkWithBall*, these are the fastest walking speeds as possible.
The tool *CustomizeOdometry* shown in Figure 7.7 works on pre-measured odometry data as a reference. The new measurements for turning and plane walks, taken on a foreign carpet, is compared to the previous ones and a factor is calculated. The data is stored on the memory stick in a file called "grab-cor.ocf" and is used as a proportional factor in the WalkingEngine to calculate the odometry. If the file does not exist or some factors are missing, the missing correction factors are set to one and the original odometry information is taken from the odometry table. To perform an adaption of the odometry, the mentioned data file must be loaded into the *CustomizeOdometry* tool. In combination with the specialized behavior *Customize-odometry*, which makes the *AIBO* perform movements with ball at maximum speed, the time for any walk can be taken by the tool. Therefore the headbutton of the *AIBO* and the corresponding "Start" button must be pressed simultaneously. The button now changes to a "Stop" button, which should be pressed, when the *AIBO* reached a previously measured point on the field or absolved a certain amount of turns. The data must be saved and written on the stick. As there are robots of different quality, this procedure needs to be done for all of them, to assure that the odometry information, especially for the older robots is correct.

7.7 Conclusion

The various efforts trying to correct the accuracy of the walks showed the complexity and diversity of this problem. Different methods are suitable for different scenarios. Adapting to special cases require a high degree of flexibility. The method of correcting *WalkReqest*s while taking measurements continuosly is capable of reaching a high precision but it is limited to correct just a small spectrum of *WalkReqest*s. As accurate as the previously measured odometry itself is the

method of inverting the odometry table. This year, a lot of time of the optimization has been spent on finetuning the *AIBO*s' walking behavior. The correction of polygonsets as described in chapter 7.6, proved to be the most effective procedure, resulting in optimized gaits

Chapter 8

Special Actions

8.1 Significance of Special Actions

“Special Actions” are in the first place used for kicks and cheering-moves. Instead of dynamically interpolating movement-polygons, joint coordinates and timings are hardcoded. The advantage of special actions is that for example kicks can be developed and tuned by hand. On the other hand are those movements not directly influenced by the environment. Special Actions are saved in “.mof”-Files. Those files include sets of joint-coordinates of each joint, timings (how many frames does the robot have to reach the destination) and an interpolation flag. The combination of those sets describe a stop-motion movement which can be “played” in appropriate situations of the game.

8.2 Creating, Editing and Using Special Actions

The mentioned “.mof”-files can be edited by a simple text editor. The body and syntax: Every set of movements has a so-called “motion_id” which has to be declared in the first line of the file. The last line has to be empty (!) and in the lines before a transition to an index file called allMotions.mof has to be set. The remaining lines between those start and end-markers can be either a set of coordinates in microradiant, timing and interpolation flag or a comment or a label or a transition to another.

Creating “.mof”-files by hand is highly unrecommended. The chapter “Tools” describes two so-called MotionDesigners, tools to create and modify “.mof”-files.

The finished file is saved in the mof-folder and the motion-id is inserted in different files to provide the capability of directly calling the movement out of the behaviour-descriptions.

In the following example the different parts of the file can be seen. The file opens with the declaration of the $motion_id$, a description (comment, started with quotes), a label to which a transition can point, the angles of the joints in microradiant (the tilde-symbol means “don’t change current angle”) and in the end of each angle-configuration the interpolation-indicator and the number of frames the robot gets to complete the motion. The file finishes with the above mentioned standard transition to $allMotions$.

```
motion_id = cheer1
"cheering after scoring a goal

label start
-32 -52 76  0 ~ ~  373 244 56 -287 12 042 -84 59 22 -651 79 17 1 25
-48 -14 9  ~ ~ ~ -68 290 23 13 55 179 -75 62 12 -540 51 57 0 50
-86 18 384  ~ ~ ~ -2 261 21 -2 210 32 -75 68 11 -608 68 68 1 50
20 895 762  ~ ~ ~ 18 3 13 -22 199 26 -51 74 17 -288 56 65 1 50

transition allMotions extern start
```

8.3 Experiences

Every developed special action works depending on the carpet and the robot. The differences between the robots are sometimes very significant. It is possible that special actions that work on one robot are completely useless on another. The same problem exists with the used carpet, so on a competition every kick has to be proved if it is working in the determined way. There are some goals that have to be achieved when developing kicks: As the joints are very weak one has to find a way to combine the strength of the joints and the gravitation in order to make it strong. The most important goal is to develop stable kicks: If kicks only work with a specified ball position and under very special circumstances they will not work in a game and have to be improved.

Another significant problem was the asymmetry of the hind legs: the right leg seemed to have a little offset of about 10 degrees in one joint. Every kick using the hind legs to gather momentum and using symmetric joint-values did not go straight forward.

The currently available kicks are listed in the appendix.

Chapter 9

Tools

For developing on the robots and a better visualization of the current status, some tools were developed and are described in this chapter.

9.1 Robot ControlXP

In the past, a lot of time time had been spent on developing tools that do not run on the AIBO platform, which is, however, very important for the development of efficient code. RobotControl is one of the most significant interfaces between the robot and the developer.

The key task of RobotControl is to show the current state of the robot in a format that is readable for the developer. That means *RobotControl* has to evaluate the data coming from the robot and send messages back from *RobotControl* to the robot. This way, the development should be accelerated by the use of efficient code. The first version of *RobotControl* was written in C++, and the basic functions for developing and debugging robot code were already avaliable. Since the number of UserControls has continously been increasing *RobotControl* hat become much too unclear over time.

A new version written in C# with the .NET Framework 1.1 had been developed. Its compiling time had became shorter. Additionally a modern programming language with a large framework was available, which simplified the development of own UserControls. The development of RobotControl2 relied on the idea to build an application that should run independently from the actual robot. That means that no robot code is contained in *RobotControl*2. Status messages are merely evaluated and indicated, as well as status changes sent to the robot. The exchange of information is achieved by the use of messages (which will be explained more detailed later) which are the basic elements of *RobotControl*2. The separation of Framework, UserControls and Managers was the main aspect in the development.

9.1.1 Motivation

RobotControl2 provided the necessary functionalities needed. The application, however was very unstable in some parts which led to system crashes. We decided to develop a new version, which

should be able to handle the errors and give us the possibility to connect to several robots at the same time. In some parts, *RobotControl*2 was very inefficient. This restricted the speed of the application.As the new .Net Framework 2.0 was available at the beginning of our project group, we decided to develop with the newest technologies in order to be able to use its innovations and above all the increased speed of the new Frameworks.

9.1.2 Fundamental structure

*RobotControl*2 was developed to create a stable framework, which could be adapted by own UserControls to our needs. This idea was taken up and extended with RobotControlXP. In the following chapters, the new structure will be presented.

9.1.3 Framework

The heart of the application is the Framework in *RobotControl*XP. This is the only relly vital part. In *RobotControl*2, the Framework was only responsible for the administration of UserControls and for storing some application values. In RobotControlXP, the Framework does the message handling as well. The following tasks are now executed by the Framework.

- Managing the connections to the robots
- Managing the messages
- Managing UserControls and Managers
- Show UserControls for interaction with the robots
- Store application values

9.1.4 Manager

To analyze incoming messages from the robots centrally and to remit them to other Managers or UserControls is the main application of the Managers. Therefore, the Managers run in their own background threads. Time-consuming calculation processes run in the background as well, this way they do not disturb or interrupt the user interface. Another advantage is that incoming messages are only processed once, and the result can be used by other UserControls and Managers respectively.

9.1.5 UserControls

The UserControls are responsible for the representation of information from the robots. There are many UserControls, and each of them has its own application. They access information from the Managers or messages from robots. Because the UserControls run in the main thread, they should only exchange information with the user. The supervision of messages is only done by

the Managers. The development of new UserControls and the use of old UserControls is very easy and will be described in the appendix.

9.1.6 Messages

To understand the methods of *RobotControl*, it is necessary to understand message handling. Basically, we can say that messages are sent from the robot to the computer and from the computer to the robot. These messages are sent together with a header which describes the content of the message. The header contains 10 bytes and is built as described following

- 0. byte: general information
  - 0000000X: messagequeuestate 0= more messages; 1= last message in queue
  - 00000XX0: teamcolor 01 = red ; 10 = blue ; 11 = undefine
  - 00XXX000: playernumber: 001 = 1 ; 010 = 2 ; 011 = 3 ; 100 = 4
  - 0X000000: source 0 = from physical robot ; 1 = from simulator
- 1.-4. byte length of the message
  The length of the message's content
- 5. byte message ID
  Identifies the type of the message
- 5-9 byte time stamp
  Consecutive number

This header is part of every message. The 5. byte shows the message ID. This message ID describes how the content of the message must be evaluated. The complete list of all MessageIDs can be found in the enum in RobotControl.Messages.MessageID.

9.1.6.1 Debugging on the robot

The content of information depends on the module in the robot code from which the messages are sent to *RobotControl*. These messages are coming from the robots. The communication of the messages is handled by streams and can be used from every module. Therefore, a message queue that stores the messages adds additional information such as playernumber, teamcolor, or source if needed. The message queue can be used to send binary data, text, or raw text. The usage of the messages is shown in the following example:

$Image\ myImage;$

$myMessageQueue.out.bin \ll myImage;$
$myMessageQueue.out.finishMessage(idImage);$

$int\ i = 3;$
$myMessageQueue.out.text \ll \text{“}ab\text{“} \ll i \ll \text{“}c\text{“};$
$myMessageQueue.out.finishMessage(idText)$

$int\ i = 3;$
$myMessageQueue.out.textRaw \ll \text{“}ab\text{“} \ll i \ll \text{“}c\text{“};$
$myMessageQueue.out.finishMessage(idText)$

$int\ a, b, c, d;$
$myMessageQueue.out.bin \ll a \ll b;$
$myMessageQueue.out.bin \ll c \ll d,$
$myMessageQueue.out.bin.finishMessage(idJustNumbers);$

9.1.6.2 Extended Debug Mechanism

The extended debug mechanism - the DebugRequests - makes the work with RobotControl easier. DebugRequests are used to switch parts of source code on and off. That means they will be executed or skipped. DebugRequests are called switches, they can be asked in the source code and will return the corresponding code. All that is wrapped by macros, for easy use.

- ```
  DEBUG_RESPONSE (id, expression);
  ```

  execute the expression if the ID is enabled.

- ```
  DEBUG_RESPONE_NOT (id, expression) ;
  ```

  execute the expression if the ID is not enabled

There are also macros for easy sending of messages.

- ```
  OUTPUT (id , format, data)
  ```

For example:

- `OUTPUT(idText, text, "Could not load file " << filename);`

We have macros in the code which can send messages and messages witch control whether some action should be done. These macros are used as in the following example:

- `DEBUG_RESPONSE ("send motion data", OUTPUT (idText, Text, motionData)`

If the DebugRequest “send motion data” is active, the motionData is sent as a text message.
  ```

9.1.6.3 DebugData

The last point of message handling, the sending and receiving of DebugData makes the work with robots very fast and simple. By the usage of DebugData, all the streamable data can be changed at runtime. It is possible to send complex data types, as well, if the definition of the data is contained in the Stream Handler. That means the data types have to be made streamable. For this application, a macro is used:

- `MODIFY (id, object)`

Finally, the variable can only be used in the calling module. From RobotControl, you can modify these values at runtime.

9.1.6.4 Messages

We already know how messages are built, how they are created on the robot, and which control and manipulation possibilities we have for them. Now some some special messages will be described in detail because they are necessary to understand the Framework.

9.1.6.5 Poll (RobotControl ⇒ robot)

There is a mechanism, called polling, to show RobotControl the actual state of the robot. It is a message that is sent from RobotControl to the robot which makes the robot send a list with all currently registered DebugRequests to RobotControl.

9.1.6.6 idDebugResponse (robot ⇒ RobotControl)

The message idDebugDataResponse is one element of the list from the result of the polling. The message describes one switch and its state. The state can be activated or deactivated. It is built like this:

- Name (string - name of the switch)
- Status (bool - 0= deactivated 1=activated)

Examples:

- automated requests:cognition main finished 1
- automated requests:DrawingManager 0
- automated requests:ModuleSolutionTable 0
- automated requests:StreamSpecification 0
- Processes: Debug - printRobotStatus 0

- send representation:percepts:all percepts 1
- Buttons:press front button 0
- cognition:show fps 0
- debug data:headcontrol:p1 1

The Name of the switch can be selected. Sometimes, the name is used to build a further group of DebugRequests. The message handling should end with a special idDebugResponse message, which contains the following content as text:

- “Polling finish”

This message shows that all DebugRequests have been transferred. Because of timing problems, this message can be transmitted before the last idDebugResponse is received.

9.1.6.7 idDebugRequest (RobotControl ⇒ robot)

The idDebugRequest is a request from RobotControl to the robot. The request is built like this:

- Enable (bool - the new value of the switch)
- Once (bool - single pass)
- Name (string - the name of the switch)

Dependent on the properties the switch stays activated or deactivated, or it is activated once.

9.1.6.8 idStreamSpecification (robot ⇒ RobotControl)

The message idStreamSpecification contains the structure of the actual streamable objects of the robot - the specification. These messages are send when RobotControl issues the following request:
Automated requests:StreamSpecification

9.1.6.9 idDebugDataResponse (robot ⇒ RobotControl)

After the specification has recognized the modifiable data, we can make the robot send us a DebugData message. For this, the required DebugData must be activated with the help of the DebugRequest’s name. To activate or deactivate it we need the name of the DebugData that is in the list of DebugRequests. We can recognize DebugData by the prefix “debug data:”. Here are some examples for DebugData:

- debug data:BallLocator:KalmanLatest:Parameters
- debug data:GoalRecognizer:Hypothesis

- debug data:motion:collisionPercept
- debug data:headcontrol:mps
- debug data:headcontrol:p1
- debug data:headcontrol:p2

After activating the DebugData by using DebugRequests, they are received by RobotControl in form of idDebugDataResponses from the robot. With the help of the stream specification, we are able to analyze the DebugData.

9.1.6.10 idDebugDataChangeRequest (RobotControl ⇒ robot)

The message idDebugDataChangeRequest initiates the modified DebugData to be sent from RobotControl to the robot. It contains the name of the DebugData, a flag that describes whether the data should be changed, and the main DebugData.

9.1.6.11 idModuleSoulutionTable robot ⇒ RobotControl)

The message idSolutionTable contains all modules that are available on the robot. In order to receive this message, a DebugRequest “automated requests:ModuleSolutionTable” has to be requested. This message has the following composition:

- ModuleName (string - name of the module’s group)
- NumberOfSolutions (int - count of modules)
  - ModuleSolutionName (string - name of the module (multiple items possible))
- DefaultSolution (string - name of the default module)
- SelectedSolution (string - name of the selected module)
- moduleCategoryName (string - name of the category)

9.1.7 Framework

The application of RobotControlXP is represented by the Framework which administrates the messages, the Managers and the UserControls. It uses the WeifenLue.DockPanelSuite[1], instead of the TD.SandDock dock panel for the visual integration of the UserControls.

The Framework consists of one main window into which UserControls can be added arbitrarily. With the help of the WeifenLue.DockPanelSuite, they can be distributed arbitrarily. The Framework contains the controller for the connections to the robot. They can be controlled by the Framework. New usercontrols have to be derived from the basic

[1]http://www.codeproject.com/cs/miscctrl/DockManager.asp

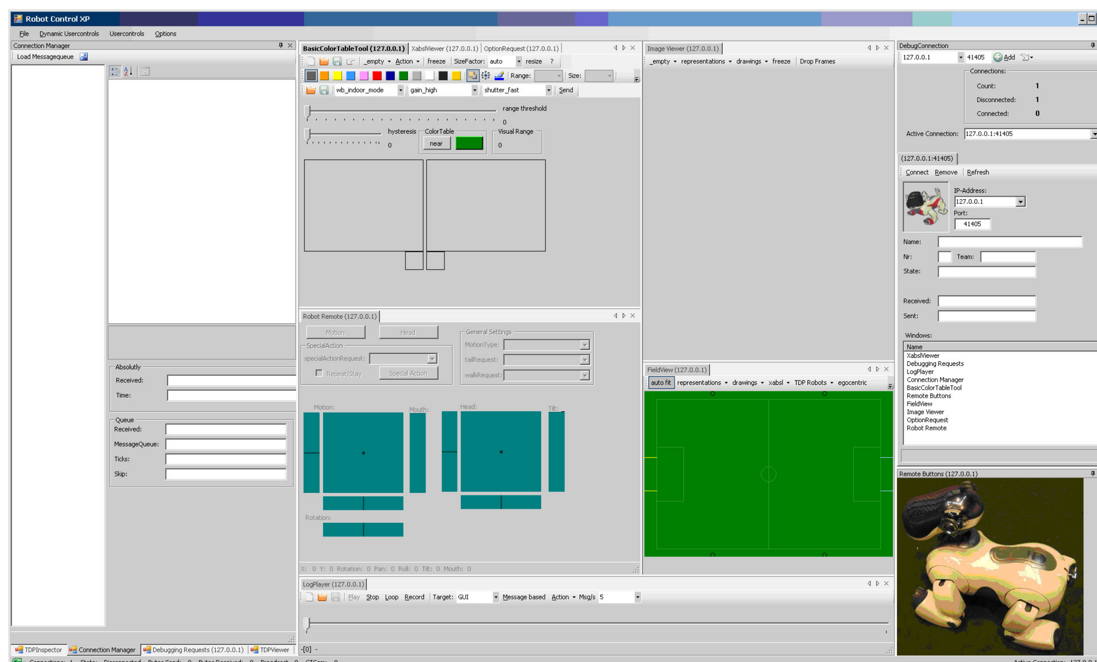


Figure 9.1: RobotControlXP with some open UserControls

`class RobotControlXPUserControl`. The basic class provide the needed information that any usercontrol can be integrated into the application. The Framework loads all available UserControls automatically and checks their states by starting the application. We differentiate between ConnectionUserControls and ControllerUserControls.

9.1.7.1 ConnectionUserControls

A ConnectionUserControl is a control which is used for a single connection. It receives data from a connection and displays it. You can open an own UserControl for each connection.

9.1.7.2 ControllerUserControl

A ControllerUserControl can use multiple connections at the same time. It can be opened only once and is then valid for all connections.

If it is a ConnectionUserControl, it follows an entry in “Dynamic UserControls”. If it is a ControllerUserControl, it follows an entry in “UserControls”.

There is the basic class RobotControlManager which integrates the Managers in the Framework and lets other Managers or UserControls use their functionalities. We differentiate between ControllerManagers and ConnectionManagers.

- ControllerManager
  Manager is valid for all connections

- ConnectionManager
  Manager that can be built for every connection

9.1.7.3 DebugConnectionController

The DebugConnectionController handles the connections. It contains a list of all DebugConnectionObjects, therefore it is also the central place for building new connections. The controller is available for every base class. It creates events when connections are added, modified, or deleted.

9.1.7.4 DebugConnectionObject

The DebugConnectionObject involves all Managers that are needed for each of the connections. Entities of the Managers are only created if they are called first. In addition, the DebugConnectionObject contains a list of all UserControls that are built dynamically by the Framework and added to the DebugConnectionObject.

Each DebugConnectionObject does the message handling on its own. It generates a new queue which then creates an own message queue for each message ID. The big advantage of this is that messages can be processed parallelly and they control each message queue. You can define messages to be skipped or to stay in the queue in order to be processed one after another. The message queue trigger an event if a message comes in. Messages to Managers or to UserControls are distinguished between. UserControls issue an invoke which is used by the main thread. This guarantees that no crash occures due to simultaneous access.

On the lowest level, there are the connections which show the connection to the robot. The Connection links by TCP/IP straight to robot. The communication runs via byte streams which are written or read in the connection. The message handler converts the message object to a byte stream or the other way around. The main message queue is in the message handler which has to generate and administrate the message queue.
Managers must be derived from the base class RobotControlManager. In order to let a Manager react on messages from the robots, the method InitManager() must be overwritten. This is achieved with the help of the DebugConnectionObject which is in the base class. The method getMessageQueue(MessageID) returns the message queue which contains an event. This event has to be bound to a method with a pointer in order to process the message.

```
GetMessageQueue(MessageID).MessageReceived +=
new MessageReceivedDelegate(HandleMessage);
```

9.1.8 UserControls

The UserControls that are used in RobotControlXP have to be derived from the base `class RobotControlXPUserControl`. This way, the Framework is able to administrate UserControls, and they can call information over the base class. The following information can be set in the base class.

- ControlType
  - ConnectionUserControl
    appoints that this usercontrol can run for each connection once
  - ControllerUserControl
    appoint that this usercontrol can run once
- Group
  the folder in the menu with which in the menu item is classified
- Image
  the picture that is show by the menu item
- MenuItemName
  the text of the menu item
- DisplayName
  the name of the form
- DefaultDock
  the default dock position
- DockAreas
  the possible dock areas
- Description
  the description of the functionality of the UserControl
- EnabledUserControl
  appoints whether the UserControl is available

To connect the UserControls with the Managers, `InitUsercontrol()` or `InitController()` must be used by the DebugConnectionObject from the base class to get the events of the other Managers.

The figure (9.1.7) shows RobotoControlXP with some open UserControls. If many UserControls and connections are opened, a mechanism was used that can show the name and the picture of one robot in order not to lose the overview. Additionally, the title and the name of the robot is shown in each UserControl. If these are not available, the IP-address is shown. The name and the picture of the robot can be chosen individually; both are stored in a xml-file in the config folder of the application.

9.1.9 Debug Connection UserControl

To build a connection with the DebugConnection, a UserControl is needed. With the add button, you can add as many connections as you like. They can be administrated in the tabs. In the tabs, all information about the connections is displayed, and you have the chance to connect to or

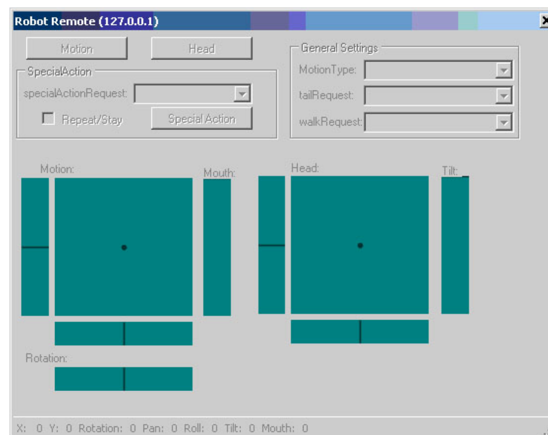


Figure 9.2: RobotRemote

disconnect from the robot or to remove the tab completely. It is important to know that during the runtime only one active connection exists. UserControls can be added by the menu “dynamic UserControls” and their status is shown in the status bar. You can change the active connection by the box “active connection” in the DebugConnection UserControl or by the box in the left corner of the window.

Additionally, active and divided connections as well as the picture of the active robot are shown in the DebugConnection UserControl.

On the Tab of the connection, you find a list of all windows that belong to this connection. You can activate them and bring them to the foreground by a doubleclicking on the list item.

9.1.10 Robot Remote UserControl

Robot remote is a new UserControl with which the user are able to operate the robot remotely by the mouse. The figure (9.1.10) shows the RobotRemote usercontrol. For this, the user have to click into the green motion field in the RobotRemote usercontrol with the mouse, and then the robot moves relative to the center point. Therefore, the DebugDataMotionRequest and HeadMotionRequest are used which are analyzed by starting the UserControl. With the dropDownBoxes, you can choose different special actions and execute them once or repeatedly by activating the corresponding check box. The head or the mouth can be remote-controlled with the RobotRemote UserControl.

There is an extra field for rotation with the robot, but you can also use the right mouse button in the other motion field to rotate the robot. It is possible to move very flexibly with the robot on the soccer field by using the apropriate controls of this tool.

Beside the special actions, there is a setting for the motion type, which automatically set to walk ‘when’ you make movements in the motion field. The tailRequest sets the movement of the tail, and the walkRequest changes the type of the movement.

Figure 9.3: FieldView

9.2 PotentialFields

At the moment potential fields are only used for positioning the robot if it is in the supporter role. Since *RobotControl* 2 it is not possible to visualize the potential fields anymore and because of this it's hard to make changes and evaluate the effects. Because of this, one new visualization and one user interface for sending fields to the robot was written.

9.2.1 New functionality of FieldView

For visualization of potential field the UserControl FieldView was extended.

9.2.1.1 Changes

The user interface FieldView (see Figure 9.3) already exists and provides information from the robot in global coordinates. The potential field is a global function with motion commands for every point on the field[11]. Because of this, the visualization of potential fields was also included in FieldView.

For activation the button "Potentialfields" was added, whose text changes to the visualized field type if the button is pressed. Available states are:

- Potentialfields
- Energyfield
- Directionfield

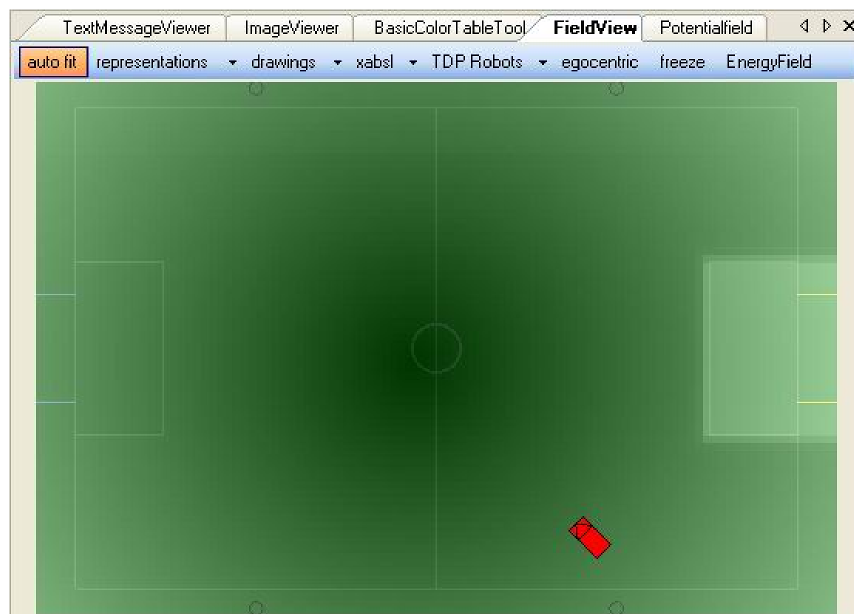


Figure 9.4: PotentialField with move-to-center and avoid penalty-area

Whenever another state than "Potentialfields" is active and a behavior with potential fields is executed, every tenth field is sent in a debug response and shown by FieldView.

9.2.1.2 Energyfield

This is the first active mode, which visualizes the energy field with a hard coded resolution depending on the window size, with an area of 5x5 pixels (see figure: 9.4). The requested resolution is dynamically sent by debug messages.

A dark color means lower areas and lighter color represents higher points. The Robot will always try to reach lower areas, which means walking to darker positions. If there is more than one point on the field sourrounded by higher areas only the robot will walk to one of them an stay at one local minima.

With energy fields it is also hard to find plateaus, i. e. areas where the energy level is constant. The gradient at this point is zero, so the robot wouldn't move if it stands there.

9.2.1.3 Directionfield

As already mentioned it's hard to find local minima and determine the motion direction in energy fields. As one solution the gradient field only shows gradients in some positions on the field, and visualizes it by arrows, showing to the direction the robot would move to. The length of gradients is neither visualized nor sent by the robot because of limited bandwidth. This information could also be seen in energy fields. Points in the grid where an arrow should be, but doesn't appear, are points without any motion command, this is a local minimum.

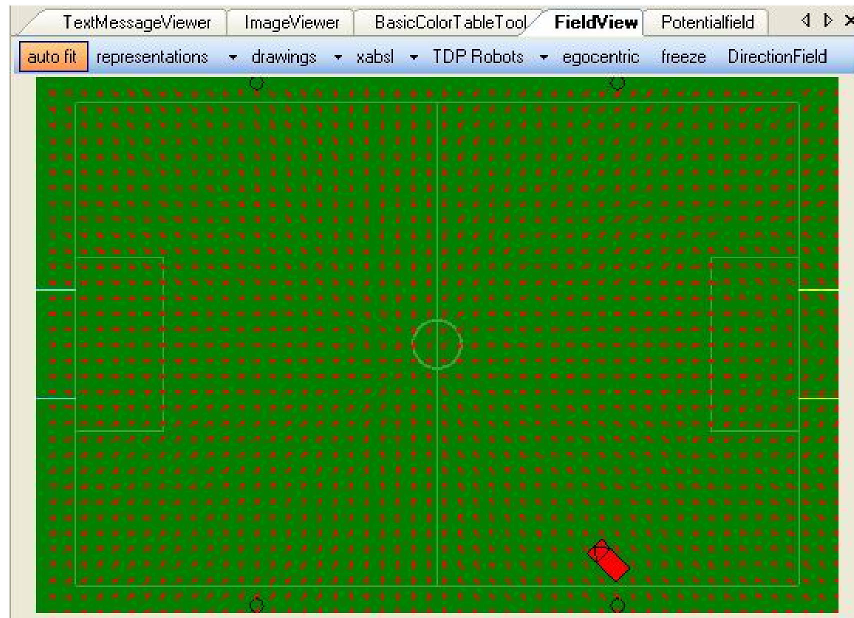


Figure 9.5: PotentialField with move-to-center and avoid penalty-area

9.2.2 Description of UserControl Potentialfield

With the UserControl Potentialfield it's possible to edit the XML-Sources of potential fields, compile them and send the compiled code to the robot.

9.2.2.1 User Interface

The user interface consists of two main parts (see Figure 9.6). In the main text box it is possible to edit the potential field source file in the same format as described in the Diploma-Thesis from Tim Laue [11]. The menu above is for creating an empty file, to load, save or undo the last steps and compiling potential fields and send the last saved one to the robot, if it's in the correct hard coded directory.

9.2.2.2 Compile and send PotentialField to Robot

With this button the Makefile under "$GT2005\Src\Modules\BehaviorControl\-MSH2006BehaviorControl\PotentialFields" is executed, which creates the compiled ".pfc"-file under "$GT2005\Config\Pfield\MSH2006". Afterwards, if makefile was successful, this pfc-File is send to the robot. If something is missing, it could be helpful to execute the makefile manually and see which potential field was wrong.

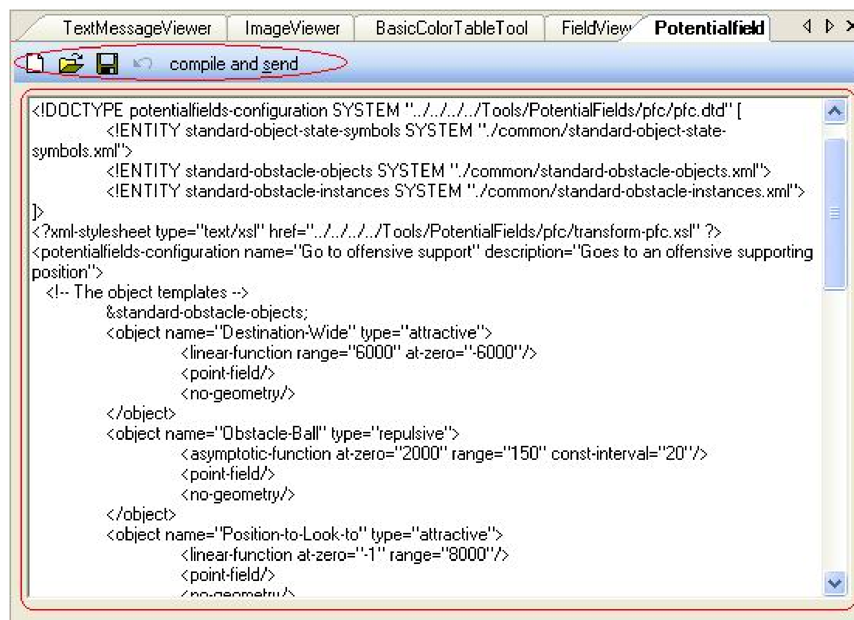


Figure 9.6: PotentialField-Userinterface

9.3 BasicColorTableTool

Since the robot uses a new color table format with two color tables, (*colorTable128*) some changes to the BasicColorTableTool become necessary. Additionally, some new hotkey functions were added to the BasicColorTableTool for a more comfortable work on color tables.

9.3.1 New functionalities

Figure 9.3.1 shows the BasicColorTableTool with some new functionalities. First we have the *rangeThreshold* slider to set a *rangeThreshold* value (1). On the right side of this slider the selected value is shown (4). A useful value is about 250 mm. Beneath we find the slider for the hysteresis value (2). Mostly no hysteresis was used, so that a value of zero is recommend. The value in the *visualRange*-panel (5) displays the current *visualRange* of the robot. The two buttons in the ColorTable-panel (6) have two functions: they show which color table (near or far) is active at the moment and by pushing one of these buttons you can manually select one color table. Every time a new image arrives, the *visualRange* is calculated and compared to the *rangeThreshold*. If the current *visualRange* is smaller than the *rangeThreshold*, the near color table becoms active. That means the near color table was used to generate the color classified image (the right one) and the red near button flashes up. Otherwise the far color table is used and the far button flashes up. All operations on the color table affect only the active one. The first step in creating a new color table should be to set a *rangeThreshold* value.

As default, the BasicColorTableTool creates color tables in the *colorTable128* format. But you can also save a color table in the old format (*colorTable64*). The currently active color table

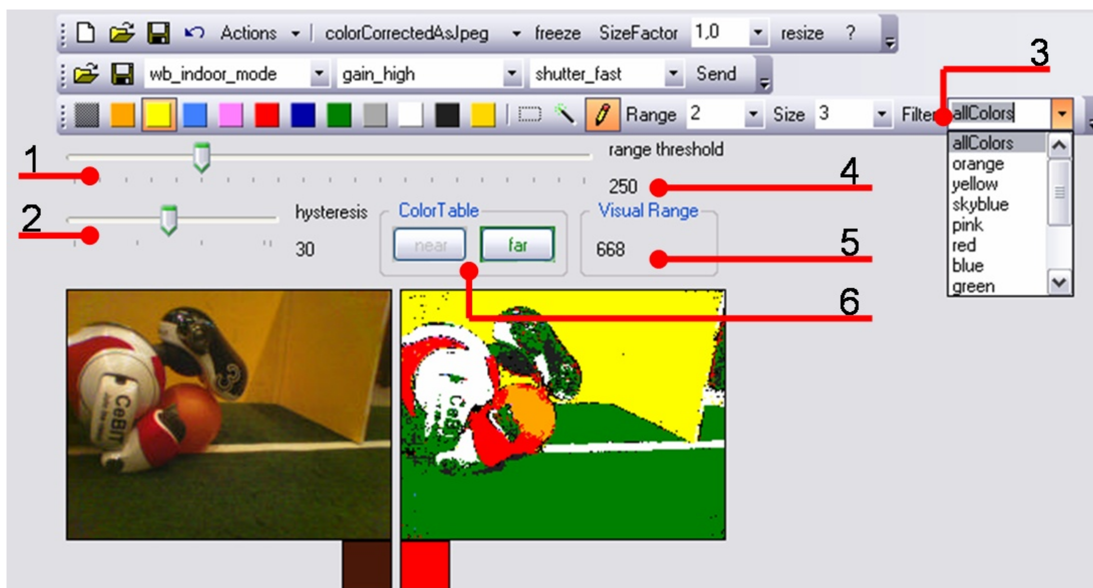


Figure 9.7: New functionalities in the BasicColorTableTool: 1) *rangeThreshold* slider 2) Hysteresis slider 3) Filter combo box 4) Current *rangeThreshold* 5) Current *visualRange* 6) Buttons to show and select active color table (near or far)

will be saved into a *.c64 file, just by selecting the *.c64 file format in the save dialog. You can also load color tables in the *colorTable64* format into the currently active color table just by choosing a *.c64 file in the load dialogue.
Using the generalization function (Menu: *Action –> Global generalization*) only affects the active color table. Thus, independent generalization of each color table included in the *colorTable128* format is possible. If you like, you can also copy each color table into the other (Menu: *Action –> CopyColorTable*). This can be useful to take a ready *farColorTable* as starting point for making a *nearColorTable*. Not starting from scratch can save a lot of time. If you decide to send the color table directly to the robot, you have to choose either the *colorTable128* or *colorTable64* format (Menu: *Action –> Send to robot*).

The BasicColorTableTool also has a filtering function now. In the filter combo box (3), you can choose which color class you want to see in the color classified image. If you select one color class for the filter, all operations only affect pixels of the chosen color class. This filter function can be very useful to check if there are, for instance, orange classified pixels where no orange should be (find potential ghost balls in red robots and yellow goals, see Figure 9.3.1). To deactivate the filtering function just select the value ‘allColors’ and all color classes are shown.

9.3.2 New hotkey functions

To make the work on color tables more comfortable and faster, some hotkey functions were added. Now it is possible to create color tables without moving the mouse out of the edit

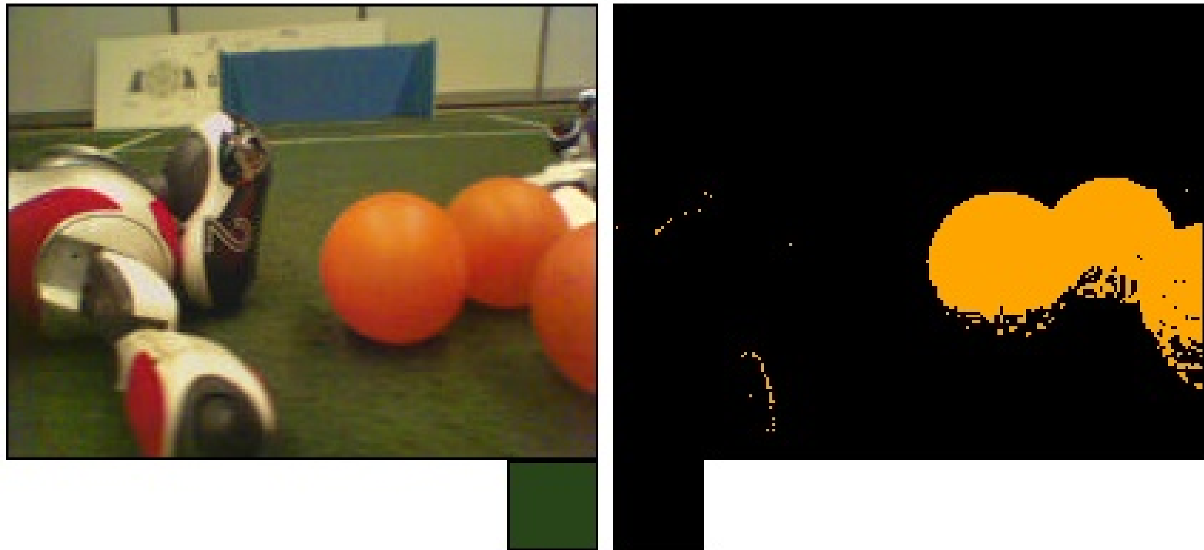


Figure 9.8: Left: original image. Right: Filter is active with the color class orange

windows. This way for instance, the mouse movements to the tool bar and back are no longer necessary. The avoidance of one of these movements only saves a little amount of time, but during the complete creation process, hundreds of these movements will be done and a lot of time can be saved in the sum. Information about the hotkey functions is also available in the BasicColorTableTool by clicking the '?' in the toolbar.

Overview of the hotkey functions:

- Y: Previous log image (useful, if you make color tables from log files)
- X: Next log image
- 1: Choose Rectangle-Tool
- 2: Choose Floodfill-Tool
- 3: Choose Draw-Tool
- C + MouseWheel: change current colorclass (the active color button in the toolbar)
- F + MouseWheel: change current filter (in the filter combo box)
- R + MouseWheel: change current range (parameter for floodfill tool)
- S + MouseWheel: change current size (parameter for floodfill and draw tool)
- Backspace: Undo function

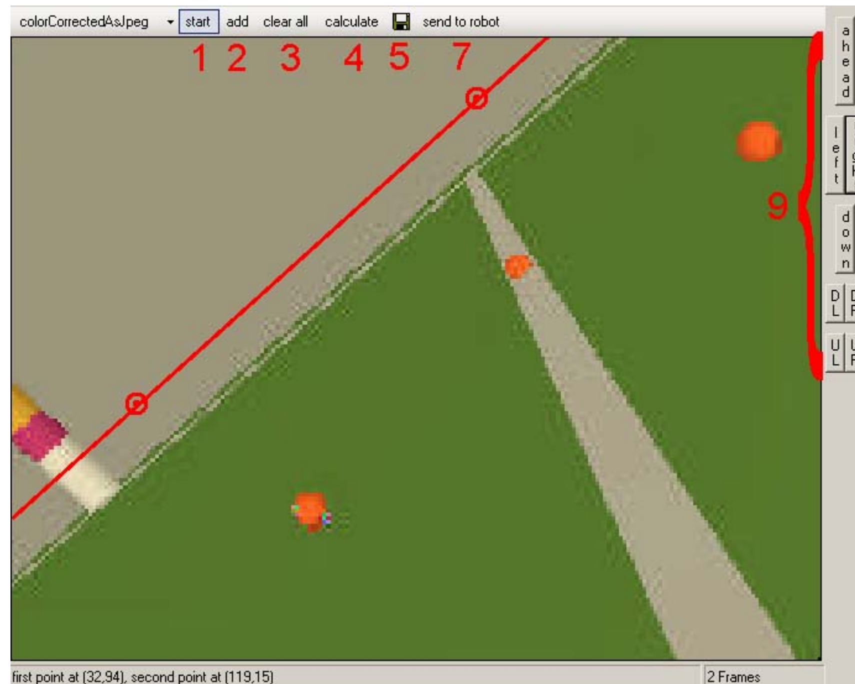


Figure 9.9: Horizon Based Tool

9.4 HorizonCalibrationTool

The *HorizonCalibrationTool* is used to find the eight calibration values, used to correct the robot's camera matrix and the horizon line (cf. Section 3). It exists in two different versions, one for the correction based on the real horizon (cf. Section 3.2.3.1), and the other for the reference-point-based method (Section 3.2.3.2). Aside from the input method (i. e. adding a frame) the two tools are similar.

9.4.1 Adding a Frame

Since adding a frame is the only significant way in which the both tools are different, it will be described seperately for both versions.

9.4.1.1 Tool Nr. 1: Horizon Based

The challenge here is to find an appropriate means to find the real horizon. One way this can be done is using a computer with a software that is capable of drawing lines. The robot is then placed before the screen and looks at a line on it. The robot is now moved back and forth. If the line as seen in the image in *RobotControl* doesn't move anymore, this is the horizon line. If it still moves, the height of the line needs to be adjusted. To start the calibration, the **start button** (1) needs to be pressed. Now the robot's joint values are sent to *RobotControl* frequently. At first the robot looks straight ahead. After the horizon has been found in this position it is marked in

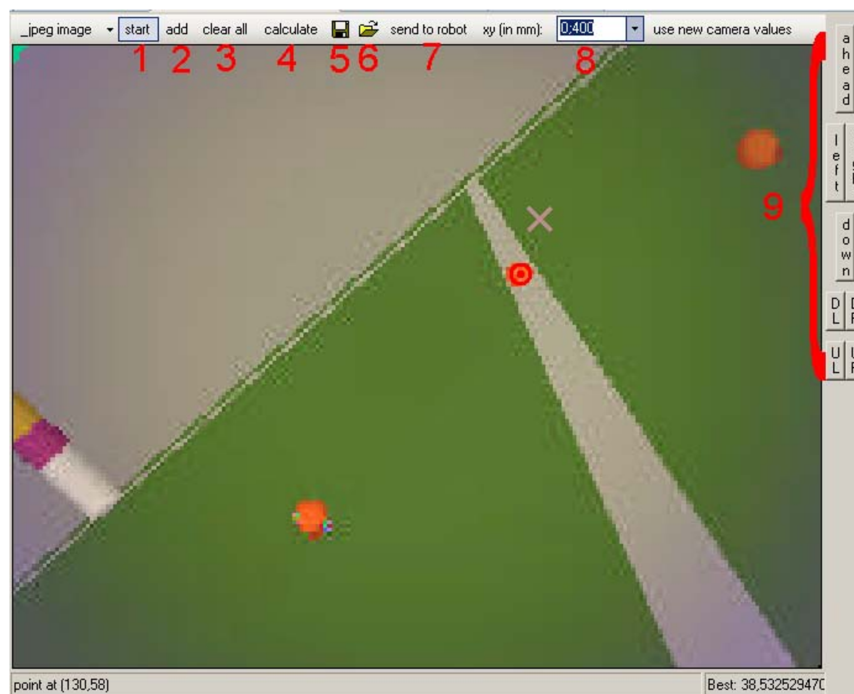


Figure 9.10: Reference Point Based Tool

the tool by clicking on two different locations on the line that represents the horizon. When the horizon has been marked, the **add button** (2) must be pressed to store the current frame. These steps are then repeated for the other head positions (*catch-ball-high*, *look-left* and *look-right*). When all frames have been gathered, one can press the **calculate button** (4) to start the algorithm that calculates the calibration values.

9.4.1.2 Tool Nr. 2: Reference Point Based

The robot is first placed in a calibration environment with specific points being marked (Figure 3.4). Again the start button needs to be pressed in order to start gathering joint values. At first the robot looks straight ahead. Then the world coordinates of the first point are selected in the combo box (8). If other positions than the provided ones are used, they can be entered manually. Then the chosen position is marked at the point where it appears in the image. As additional information the calculated position of this point is also marked in the image. This can later be used to test how good or bad a set of calibration parameters is. After the point has been successfully marked, the **add button** (2) must be pressed in order to store this frame. This procedure is then repeated with the other visible points, and continued with the other available head positions.

9.4.2 Other Functions

- The calibration depends on the robot's joint values. Since it is not desired for those values to be sent to *RobotControl* all the time, selecting the **start button** (1) starts gathering those values. If the button is deselected, the values aren't sent anymore.
- The **clear all button** (3) removes all stored frames, and calibration can be started from the top.
- If all frames a gathered, the **calculate button** (4) can be pressed. Now you can lean back a while and wait while the calibration values are being calculated. When the calculation is finished, a message box with information about the quality of the values appears.
- After the values have been calculated, they need to be saved to a text document with the **save button** (5). Now they can be inserted into the 'robot.cfg' file in the 'Config' directory under the name of the respective dog, and its horizon is corrected with these values from the next startup.
- If one wants to test the values immediately they can also be sent to the robot directly via the **send to robot button** (7).
- The tool used for the reference-point-based method additionally has a **load button** (6) which can be used to load an existing set of calibration values, which could be sent to the robot, e. g. for testing purposes. The loaded file must have the same format as a saved file would have.
- It has been found sufficient to use four different head positions when doint a calibration: *look-straight-ahead*, *catch-ball-high*, *look-left* and *look-right*. The **head-position buttons** (9) are a convenient method to let the robot look in different directions. The top four head positions are used for calibrating, while the lower four can be used additionally to test a set of parameters.

9.5 ImageViewer

Among other things, the ImageViewer is used for different visualizations of debug information. Two new visualization functions were added.

9.5.1 Motivation

Along the scanlines of the main scan grid of the ImageProcessor, state machines are used to determine field lines and obstacles. One key idea in that approach is to use gradient information of the luminance (brightness) or the chrominance (color) channel of an image. For example a positive gradient that is greater than a adequate threshold in the Y-channel could indicate a transition from the dark green carpet to a bright field line. Thresholds for the different types of gradients are stored in the ImageProcessor.

Different lighting conditions make it necessary to create new color tables. But also the thresholds should be adapted. Smaller threshold values tend to produce noisy results (see Figure 9.11(d)). Too large thresholds could lead to ignoring existing edges. Two new functions in the ImageViewer, explained in the next section, will help to find adequate values for all channel thresholds.

9.5.2 New functionalities in the ImageViewer

The ImageViewer has two new functionalities: *gradientNeg* and the *gradientPos*. These functions show a grayscale image with the main scan grid of the ImageProcessor (in black) and some colored pixels indicating gradients greater than their corresponding channel thresholds (see Figure 9.11(d)).

9.5.2.1 Gradient color code

To distinguish between all combinations of gradients, we make use of a color code like the one shown in Figure 9.11(b). The Y-channel is the luminance, the U-channel the chrominance difference to red and the V-channel the chrominance difference to blue.
The folliwing list shows the mapping of colors to gradient combinations:

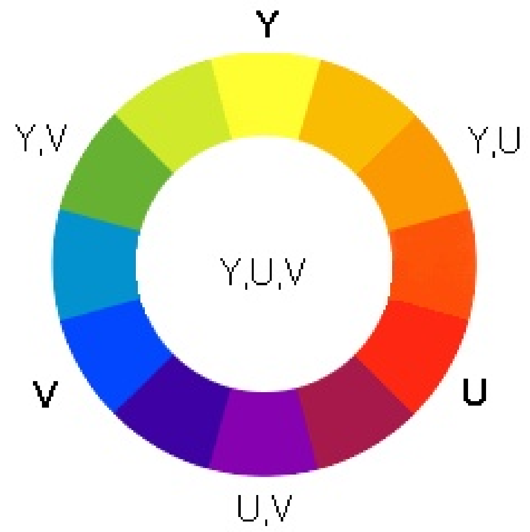
- yellow: Y gradient
- red: U gradient
- skyblue: V gradient
- orange: Y & U gradient
- green: Y & V gradient
- pink: U & V gradient
- white: Y & U & V gradient

The resulting color of a combined gradient is comparable to the resulting color after mixing up colors in a paint box. If you, for instance, mix up yellow and red, it results in orange. Thus orange is the color indicating a Y-channel and a U-channel gradient combination.

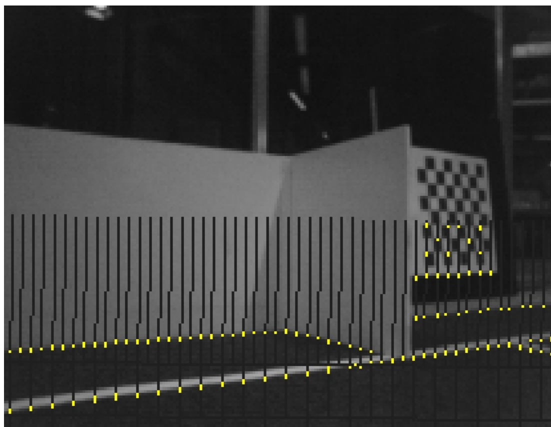
9.5.2.2 Channel threshold calibration

The *gradientNeg* function visualizes negative gradients while *gradientPos* visualizes positive gradients. Using the *DebugDataUC* dialogue (choose *gradientThresholds*), all thresholds can be modified. The result of the modifications can be evaluated on the fly. The user has to decide whether the current values are good enough for the existing lighting conditions. On the one hand, all relevant edges should be found and colored by using the gradient color code. On the other hand, no ghost-edges should be indicated by using too small threshold values. Finding a good

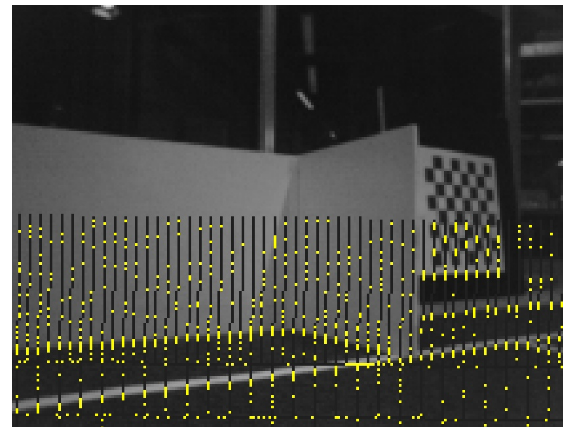
(a) Original image



(b) Color code to indicate a gradient combination



(c) Negative gradient image with adequate threshold for Y-Channel (here 20)



(d) Negative gradient image with too small threshold for Y-Channel (here 3)

Figure 9.11: Negative gradient images (*gradientNeg*) with different thresholds for the Y-Channel.

balance is the main approach of the threshold calibration. Well calibrated threshold values can lead to more accurate results of the state machines in the ImageProcessor and to more reliable percepts. So far the thresholds are hard coded in the ImageProcessor, but it would be only little work to make them variable and to store them, for instance, in the color table file.

9.6 Motion Designer

The MotionDesigner was developed to easily create and edit special actions. With the tool it is possible to concentrate on the real work of developing and editing stop-motion-moves instead of being concerned about the syntax of ".mof"-files. Beside the usual features of loading, saving and creating new files there are some special features that have to be explained.

9.6.1 The Tool(s)

The tool itself exists in two different versions depending on the main program (RC2 or RCXP). Both tools have mostly the same features, the RXCP-version has some additional helpers partially made possible by the .NET 2.0 framework.

9.6.1.1 The Main View

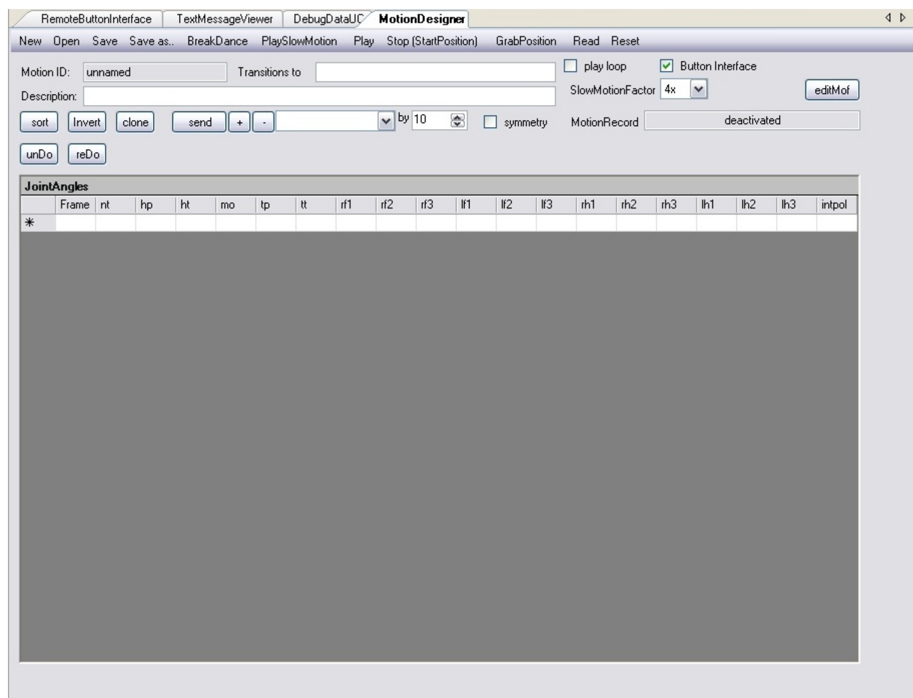


Figure 9.12: Motion Designer (RC2)

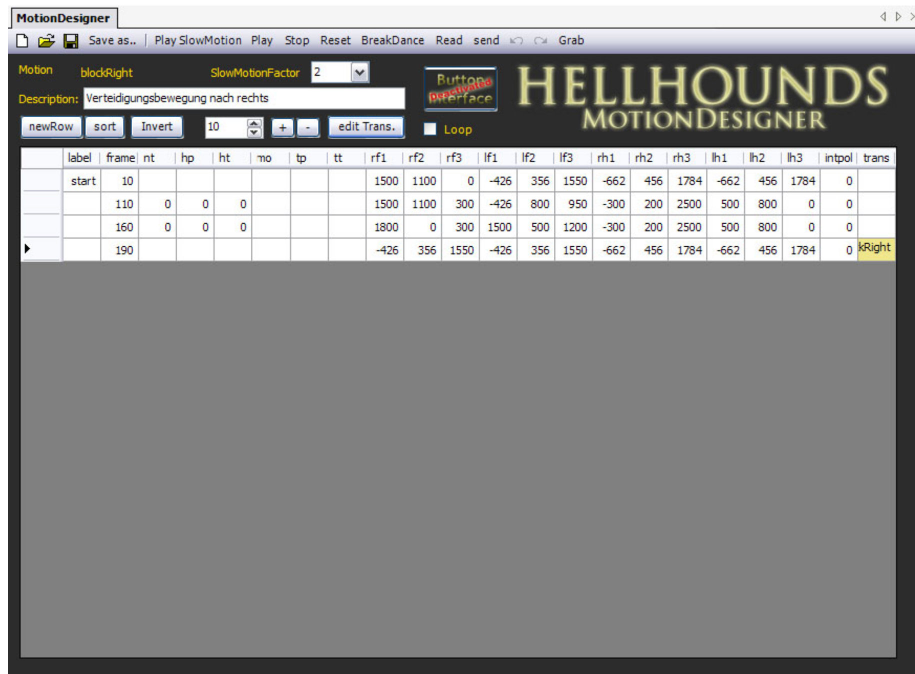


Figure 9.13: Motion Designer (RCXP)

The view in figure 9.12 and figure 9.13 shows different parameters for ".mof"-files. After loading an existing special action the motion-id is shown, descriptions from the file are parsed and the chain of movements is visible in the table below.

Joint-angles in the table can be directly edited as well as the frame numbers and interpolation flags. As already mentioned the motion can be seen as a stop-motion-movement. This term is derived from older cinema-films and meant that every position of an object in a film was once photographed for one frame of the film. The motion-framework of the AIBO works at about 125 frames per second. The difference between two frames is the time the joints have to reach the defined position in point-to-point-movement (interpolation-flag set to 0) or using linear interpolation (interpolation-flag set to 1).

9.6.1.2 Loading and Saving a ".mof"-File

The program remembers the directory of the last used ".mof"-file. Loading and saving a ".mof"-file is possible by a standard-windows-dialog. When saving a file using a new filename (save as...) the motion-id and transitions (explained in "creating a mof") will be automatically set.

9.6.1.3 Editing a Special Action

Elements of ".mof"-files. A ".mof"-file consists of a few elements: the motion-id, descriptions, joint-angles, timings, interpolation flags, labels and transitions.

Transitions. Transitions are used to combine special actions or to stay in a special action. For using transitions a label has to be set where the transition should point to. After setting this label a transition from a file to another or to itself and the target-label can be chosen.

Using the Motion Designer. Once a file is opened there are some possibilities to form a new special action. The dispersion-mode, entering and directly changing joint-values or a mixture of both. In order to directly see changes on the robot (for the dispersion-mode indispensable) a connection to the robot has to be established and the motion-solution of the SolutionSelection-view has to be set to "debug". After this the "read"-command causes the program to fill the read out joint-angles of the robot into a new line of the table. Additionally the robot reacts to the "send"-command: selected lines (or in RCXP even cells of lines) will be sent to the robot and the robot will try to achieve those positions.

9.6.1.4 Execute Functions

If you have successfully opened or created a mof, you can behold it on the robot by pushing the "Play" button. If you wish to repeat this motion automatically, select the"(play) loop" click box. "Stop" sets the robot to its initial position and "Grab" in RCXP ,or alternatively "GrabPosition" in RC2, sets it to the initial position for grabbed kicks.

9.6.1.5 Additional Functions

clone (only in RC2). The function "clone" really clones one selected row. Inclusive the frame. Hence, the frame must be changed manually, otherwise the .mof will not be executable.

sort. Sorts the rows in regard to the frame numbers.

send. Sends selected lines to the robot in order to view the joint-positions directly on the AIBO.

plus and minus in RC2. First of all a joint has to be selected in the drop down box right next to the plus and minus buttons. And secondly a number of degrees must be selected right aside. After that, the selected joint can be moved stepwise by pushing the plus or minus button.

invert. Mirrors the right and left side of the movement. A left-kick can easily be converted to a right kick using this button. (Never relay only the resulting joint-angles: the hind right leg might need a correction afterwards as it has a little positioning error in one joint.)

9.6.1.6 Window "EditMof" (only RC2)

This window (figure 9.14) can be opened by clicking the button "EditMof". This tool is simply a text editor. So it is suggestive to use this tool to add some loops and other text information to the final ".mof"-file.

If it is desired to expand a final ".mof", then the window should be opened by clicking the button. There will appear a text window with values in the same order (beginning with nt, hp, ht, mo, ...) like in the MotionDesigner. Excepting the last values are the execution duration time. After finishing the changes the "Save" button can be pushed. A new window appears. Here, the directory and the filename to save the expanded ".mof" must be entered.

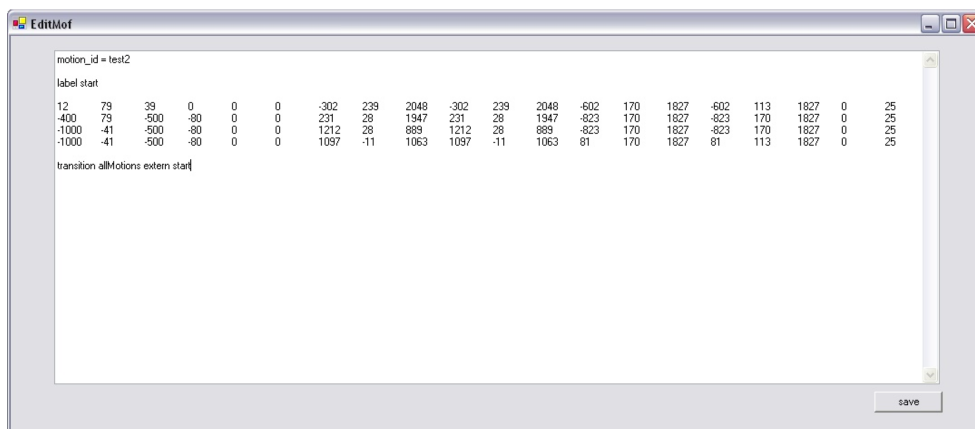


Figure 9.14: "EditMof"-View (RC2)

9.6.1.7 The Transition Editor (only RCXP)

The Transition Editor (figure 9.15) has the purpose to easily edit transitions inside the current file as well as define transition to other special actions. It can be accessed by selecting the line after that the transition should take place and pushing the "edit Trans."-button. Once the editor is visible, the active transitions of the selected line are listed in the lower part of the window. In the upper part new transitions can be chosen and added to the list (never forget to add the transition before closing the window!). There are two predefined sets of transitions which can be selected by pushing the "standard"-button. The standard-transition for leaving the special action and a transition that replays the current file starting from the "start"-label. (The current file is called "this" in the editor.) After choosing a target ".mof"-file from the list the included labels are shown and will be selectable.

9.6.2 Button Interface

The ButtonInterface (see figure 9.16) was developed to become more independent from the laptop or even the desktop PC. This makes it possible to use the above Execute Functions, excepting "Reset" and "BreakDance", directly with the robot.

To use the ButtonInterface, the solution "MotionRecord" under BehaviorControl in the SolutionSelection menu in RC2 or alternatively in RCXP must be selected. Ensure that the MotionControl in the SolutionSelection is set on debug. The SolutionSelection can be opened in the menu bar of RC2 or alternatively RCXP under the point "View".

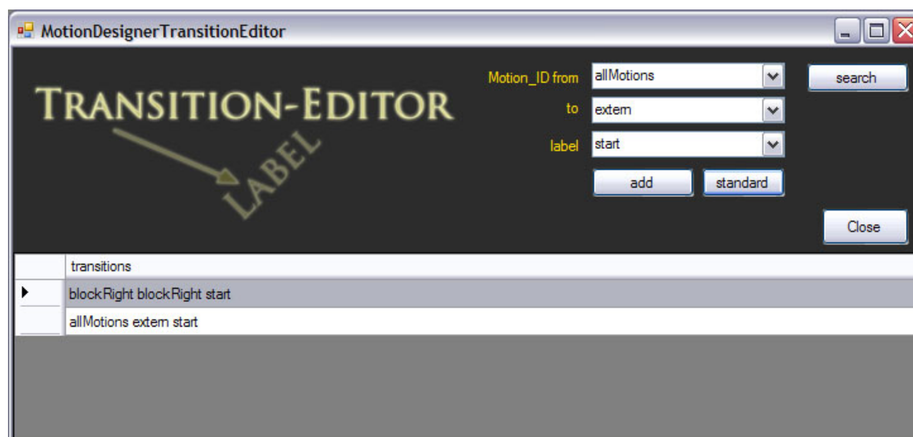


Figure 9.15: Transition Editor

By pushing the red backBack button the robot moves the joints to the “StartPosition” or stops an actual movement. Moreover, by holding the red button the robot moves to the “GrabPosition”. The yellow backMiddle button executes the actual opened mof scene in normal speed. With the blue backFront button the actual joint positions can be readout. The last white head button excecutes the mof in slow motion.

The current state (“active”, “deactivated” or “connection lost”) of the button interface is also observable in the motion designer (see figure 9.17). In RC2 it is similar.

9.6.3 Message Handling

Information about the buttons are send from MotionRecord and catched by the MotionDesigner:

```
MODIFY("MotionDesigner:Buttons", activeButton);
```

Joint values are send from DebugMotionControl and also catched by the MotionDesigner:

```
MODIFY("MotionDesigner:jointValues", jointDataBuffer.frame[0]);
```

Messages are received by the MotionDesigner:

```
public void OnConnectionEstablished( object sender,
 System.Net.IPAddress ipAddress,
 TDPRobotInfo info)
    {
      _debugDataManager.RequestAdditionalDebugData
       (this,"MotionDesigner:jointValues");
      _debugDataManager.RequestAdditionalDebugData
```

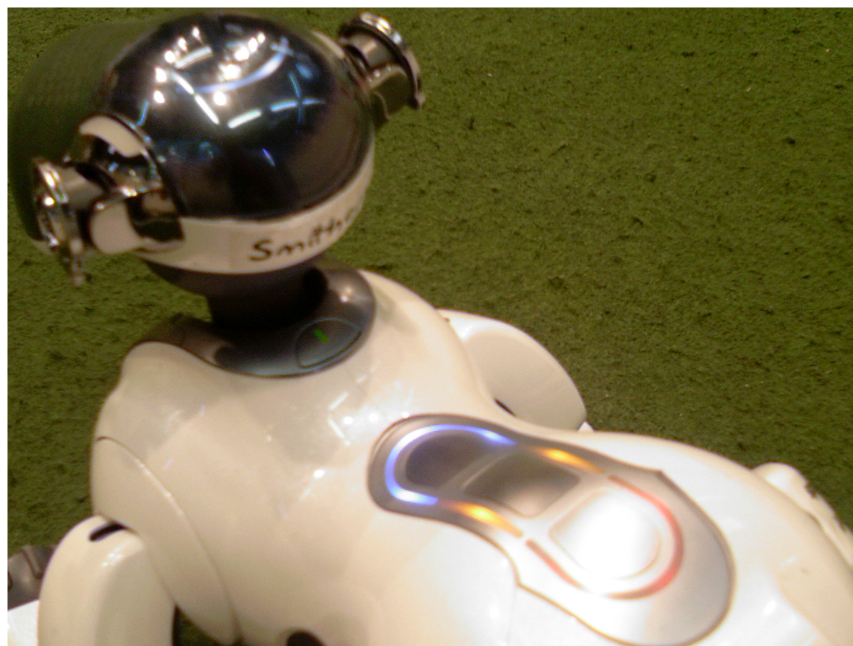


Figure 9.16: Button Interface

Figure 9.17: States of the Button Interface in RCXP

```
        (this,"MotionDesigner:Buttons");
    }


public void OnDebugDataReceived(object sender,
DebugData debugDataReceived)
    {
    if (debugDataReceived.Name.Equals("MotionDesigner:Buttons"))
    ...
    }
```

To send all values from the MotionDesigner to the Robot, a ByteArrayCreator is used:

```
RobotControl.Tools.MessageHandling.ByteArrayCreator creator =
new RobotControl.Tools.MessageHandling.ByteArrayCreator();
```

Chapter 10

Challenges

During RoboCup 2006 and the DutchOpen 2006, three challenges had to be met. These challenges are described in this chapter.

10.1 Passing Challenge

In this challenge, three robots are supposed to pass the ball between each other.

10.1.1 Challenge Rules

> This “...” challenge is intended to encourage teams to develop passing and catching skills. In this challenge, each team will be required to provide three robots, all robots must be in the same colored uniform (the decision on red or blue uniforms can be made by each team).
> Each robot will be placed on the field inside a circle of radius 30cm. The center of the circles will be no closer than 75cm and no further than 200cm apart. The triangle formed by the circles will not be equilateral, i.e. the distances between robots will be different.
> The center of each circle shall be written to each team’s memory-stick as a text file in the topmost directory: *points.cfg*. The format of the file has one target point per line, the x coordinate followed by the y coordinate. [...] The coordinates are given in cm, and the origin of the coordinate system is at the center of the field. The y-coordinates to the right of this axis are negative; on the left they are positive. Each team is responsible for writing code to read the file with circle locations.
> Initially the robots will be placed inside a circle and in the ‘set’ state for 15 seconds, this will enable them to localize. The robots will then be placed into ‘playing’ and given two minutes to pass the orange ball around.
> A pass will be regarded as successful when:
>
> - The passing robot releases the ball from inside its circle *and*

- the catching robot stops/controls the ball inside its circle. Stops/control will be left to the referees' discretion. Examples are:
  - The ball comes to a complete stop.
  - The ball is caught and held by the robot.
  - The robot is capable of hitting the ball from one circle to another without the need for stopping or grabbing the ball.

A pass will be deemed *partially* successful if:

- The passing robot releases the ball from inside its circle *and*
- the catching robot touches the ball inside the circle but the ball then travels outside the circle.

A pass is deemed unsuccessful if:

- Either robot makes contact with the ball *when* the ball is outside a circle *or*
- the ball exits the field.

A robot is deemed to be inside a circle if two legs are inside the circle. The ball is inside the circle if some part of the ball is inside the circle or on the line. That is, the line is regarded as inside the circle.
Robots may pass between each other in any order, but will be rewarded for passing to a different robot then that which passed to it.
Scoring of the challenge will be as follows:

3pts For a successful non "return" pass that directly follows a successful pass reception.

1 pt For a successful pass.

0.5 pt For a partially successful pass.

If two teams score the same number of points, the result is a draw.
All normal game rules apply in the challenge, except:

- When a ball leaves the field it will be replaced back in the closest circle.
- A robot may "ball hold" when the ball is not in the circle. This allows a robot to retrieve ball and then return to a circle to pass.

If a rule is violated then any pass resulting from this violation will receive no points. [1]

An example positioning of the robots is shown in figure 10.1.

10.1.2 Problem Analysis

For this challenge, several problems have to be solved. The biggest problem is the role assignment. The three robots should be divided into a passing robot a two receiving robots.

Figure 10.1: An example placement of the robots for the passing challenge. The circles will be drawn on the field but will not be visible to the robots.[1]

Another big problem is the ball handling. The robots have to pass the ball, receive the ball and retrieve the ball to the passing area, which leads to the third problem: the robots should be able to walk back to the passing area with the ball in case they have to bring it back into their circles. A further problem is the communication between the robots. It should be possible for a robot to say *"I am the passing robot"* or *"I am a receiving robot"* or just *"I have the ball"* or *"I have kicked the ball"*.

10.1.3 Our Approach to the Challenge

The following sections describe in detail how the challenge of passing the ball successfully between three robots was met and how we implemented our methods of resolution.

10.1.3.1 Role Assignment

We tried two approaches for the role assignment problem: a static and a dynamic one.

Hard-coded. Our first approach was to define hard-coded roles. This way it would be easier to design and test the remaining parts of our solution. Thus, one robot is defined as the passing robot and the other robots are defined as the receiving ones. This means also that the passing robot is the only one that goes to the ball. After the robot grabs the ball, it passes it to the robot next to it; that robot grabs the ball and passes it to the third robot. This Robot then passes back to the first robot (see figure 10.2).
For this approach the communication is very important. After a robot passes the ball, it has to

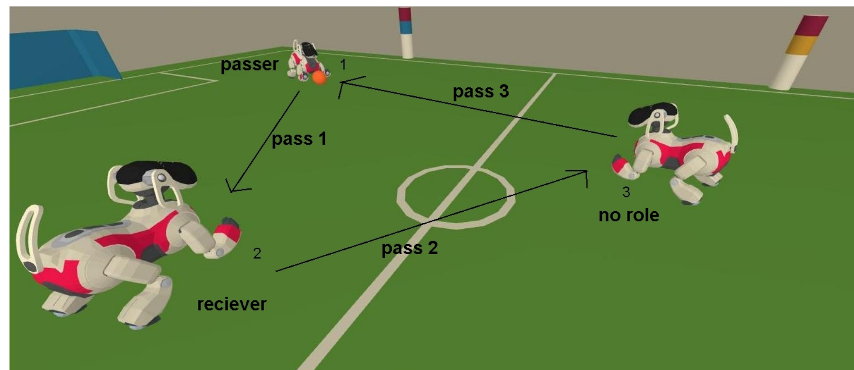


Figure 10.2: The hard-coded role assignment for the first pass

communicate that the pass was done. Then the roles have to be switched.
The problem of this approach is the communication between the robots. If the *"switch role"*-call is not received by all robots, they become confused and have problems with the further role assignment. In the worst case, the robot that is intended to become the passer will not receive the ball, but instead it will keep waiting for the passing robot (which is itself) to pass the ball. In

this case, the challenge would fail completely, because none of the robots would become passer again and so no one tries to get to the ball.
On the other hand, the advantage of this approach is its robustness. Since having a well-defined order of the robots' roles, there is no problem to figure out which robot has to go to ball. Even if a pass fails, only one robot tries to get the ball again.

Dynamic. In the rules, it is not defined which robot will get the ball first, and the exact way the ball rolls after a kick is, of course, unknown. At the beginning and after each kick, every robot looks at the ball and tries to catch it, if the ball rolls by or towards it. If a pass fails the robot closest to the ball position has to move and bring the ball back to the circle. If one robot decides to go to the ball, then it becomes the striker and all other robots have to wait for the pass. The decision which robot is nearest to the ball and how roles are dispensed is very similar to the dynamic role assignment in a game. Each robot announces its position via WLAN and calculates an estimate for the time it will take to reach the ball. To make the role assignment more stable, the current striker gets a time bonus, and each robot also gets a penalty depending on the distance between the center of its circle and the ball position.

One crucial problem of the dynamic role assignment is the ball position, because it is not a communicated, shared value, and every robot could see the ball in a different position. For the case that none of them becomes striker, the robots have to relocalize and find the ball again. The existence of two concurrent strikers is not checked.

10.1.3.2 Ball Handling

Turning. If the robot goes to the ball, it will usually not have the right direction to kick the ball directly to its teammate right away. That means, it has to turn and adjust its orientation. There are two possible ways of doing that. Either the robot grabs the ball and turns with the grabbed ball, or, the other possibility is to revolve around the ball without touching it. Because of our problems with regrabbing the ball, we decided to have the robots turn with the grabbed ball.
Nevertheless, there are two problems with this method. One is that the view of the robot is limited once it has grabbed the ball. In this case, it has to rely mostly on its odometry data. The other problem is the time limit allowed for grabbing: robots are not supposed to cover more than half of the ball with head or legs for more than three seconds, otherwise it is considered as ball holding and therefore a penalty. The turning has to be finished within these three seconds, or it has to be interrupted. In this matter, we used a very fast rotation to eliminate the problem with the time limit.

Kicking. We had to find a kick that was, on the one hand, strong enough to reach the other player, but on the other hand, slow enough so that the receiving player had a chance to grab the ball before it would bounce off. The rules [1] state that for a successful pass, the passing robot

and the receiving robot have to stay in their circles. The distance between them various between 0.75 and two meters. Because of this, existing kicks, such as *?* are not strong enough. But a "normal" front kick is too strong. So we had to create a new kick. The requirements of the kick were the right speed and a reliable kick-direction. The base of the new kick was made in the lab, but the last adjustments were made on the specific carpets at the competitions. So we got a kick that went straight and had the best speed for our concerns.

Receiving. We also had to find a way to recieve a pass. Even though a goalie needs to stop the ball as well, the blocking action has to be quite different. For the goalie, the main task is to block the ball and kick it away. In the passing challenge, it is more important to control the ball. This was one reason why the improved blocking actions of the goalie were non-applicable for our task. Another reason was that we could not be sure the ball would hit the receiving robot exactly in the middle of its chest. It would even be possible that the ball passes the receiving robot. So we created a new blocking action, for which the robot lies down quickly and spreads its front legs. But the getting up was done slowly. This way, we made sure that the ball could be controlled, and the effort to bring the ball back was minimized.

Retrieving. As mentioned before, a pass is deemed successful if both robots are in their circles at the time it is performed. But if the ball lies outside these circles, it has to be brought back in. For this task, we had two different approaches. The first approach was to grab the ball and walk back to the circle with the ball grabbed. The problem with this approach is that it takes a long time, and the probability that the robot will lose its orientation is quite large. Our second approach was to kick the ball outside the field. As it can be seen in the official rules [1], the ball is, in that case, put back into the circle nearest to the position where it has left the field. With this method, the problems with time and poor localization could be solved.
Nevertheless, we decided to take the first approach on the competitions. In tests we found out that the orientation problem was not so bad at all and the time problem could be solved by using a new walk type. With this new walk type, the robots are nearly as fast as when they are walking without the ball. And since they have to go back to circle in every case, we decided to keep the ball grabbed. Although a robot is not allowed to walk with the ball grabbed for more than 50 cm in the normal game, it is no problem in this challenge, because robots are allowed to "ball hold" for the purpose of retrieving the ball to their circle.

10.1.3.3 Communication

Position. Knowledge of the positions of the teammates is vital for successful pass play. In order to kick the ball into the right direction, the passing robot needs to know where the receiving robots are standing.
At first, we wanted to have the robots communicate their positions amongst each other, and we created a vector of position coordinates that was supposed to be filled with the position information sent by each robot in the "robot" pose part of its team message. But after the rules

for the challenge were relaxed and the positions of the robots in their circles was now known from the "points.cfg" file on their memory sticks, this approach was needlessly complicated. We only had to make sure that the robots stayed at the midpoints of their circles, and the passing robot could simply aim its kick at the designated position of its pass partner. This made the communication of the robots' positions entirely needless.

Roles. The communication of the roles (i.e. passer and receiver), of course, could not be disregarded. It showed to be quite a challenge to assign the right roles at the same time. We had to synchronize all three robots after each pass, whether successful or not, and redistribute the roles in a feasible order. As mentioned above, this was done similarly to the dynamic role assignment in game situations, but with slightly different considerations regarding the ball distance. The purpose of the role communication was mainly to avoid assigning the same role to two robots, at least in the case of the robot who is going to perform a pass.
After each role change, every robot tests which role it has been assigned and accordingly prepares for grabbing and passing the ball or for keeping its position and receiving it. In all these states, every robot constantly checks for the team message signal "pass_finished", which will trigger a role change again. The signal is sent whenever the passer kicks the ball, or one of the receivers gets the ball, or when a timeout occurs in one of the two actions. In reaction to the signal, the robots either synchronize and change their roles, or, if that fails because one of the robots did not receive the call, they go into the "reset-roles" state. In this state all old team messages are rejected and a role change is initialized.

Ball States. The three players need to know the state of the ball in order to decide how to act. If one robot sees the ball somewhere on the field, it cannot decide whether it should pass it or receive it or recapture it, unless it knows its role *and* the state of the ball. Because of that, the striker will send a team message when passing the ball, saying that a pass is being performed and the ball is ready to be stopped and controlled by the designated receiver.
Correspondingly, the receiving robot will send a team message for the other robots as soon as it is in control of the ball or when the ball takes too long to arrive or when it cannot see the ball anymore. Internally, the receiver makes use of the output symbol "ball_was_received", which activates the function "set_pass_done". Outwards, this was already mentioned as the role change trigger "pass_finished", and it means that the ball is neither being kicked nor received and that the player closest to the ball has to be reconsidered.
All team messages and xtc-symbols, including the ones related to ball states, are reset after each role change (when the new role is tested, precisely).

10.1.4 Evaluation of the Results

The outcome of our passing challenge at the world championship in Bremen showed that our approach reformulate: we managed to score three points for a successful pass and thus came off second best in this challenge.

But the approach held some problems, as well. It relied entirely on a well-working WLAN connection so the robots could communicate, but the overloaded network during the competition slowed the whole sequence down considerably. The team messages, which are checked for their age when they are needed, seemed to be invalid way too often. As a result, the robots were stuck between the role change and the reset roles states most of the time. Sometimes, the robots lost their designated roles again even though they had already started grabbing the ball or at least getting close to it.
Besides, the approach was also extremely dependent on the self-localization of the robots. The two robots, which were not assigned as being the striker currently, tried to return to their positions continuously, and whenever their self-localization was bad, they wandered off and were not in place to receive the ball when it was passed to their proper positions. Also, the striker would in case of a bad self-localization spuriously try to retrieve the ball to its circle and thus kick it from outside of its circle, therefore scoring no points.

In conclusion, the passing challenge works very well if the conditions are favorable. If vision and self-localization work well, the communication over WLAN is not constricted, and the special actions for kicking and blocking are adapted well to the prevailing conditions, then the robots are able to pass the ball around effectively and without return passes.

10.2 The New Goal Challenge

In order to increase the similarity of robotic soccer with real soccer, the appearance of the goal was changed. Testing the performance of the robots playing with these new goals was the objective of this challenge.

10.2.1 Challenge Rules

"The procedure for this challenge is similar to that of the variable lighting challenges attempted in previous years, but it use the 2006 penalty shootout rules. The team attempting the challenge places a single blue robot (robot with a blue uniform) on the field. That robot must score as many goals as it can into the yellow goal in three minutes. The team that scores the most goals wins. In addition to the single blue robot, three red opponent robots are also placed on the field. All of these robots are paused, frozen in the UNSW stance. None of them shall move during the challenge. One is placed somewhere inside the yellow goal's penalty area. The other two are placed in the half of the field containing the yellow goal, at least 30cm away from the edge. The exact locations of all the robots shall be determined by the referee, and will be the same for all teams. There is a single ball upon the field. Initially it is placed in the center kickoff position. Upon each score, the ball is moved back to the center kickoff position. The robot is not moved by the referee and must make its own way back to the center of the field to reach the ball again. The robot will have its back button pressed when the ball is moved back to the center to indicate

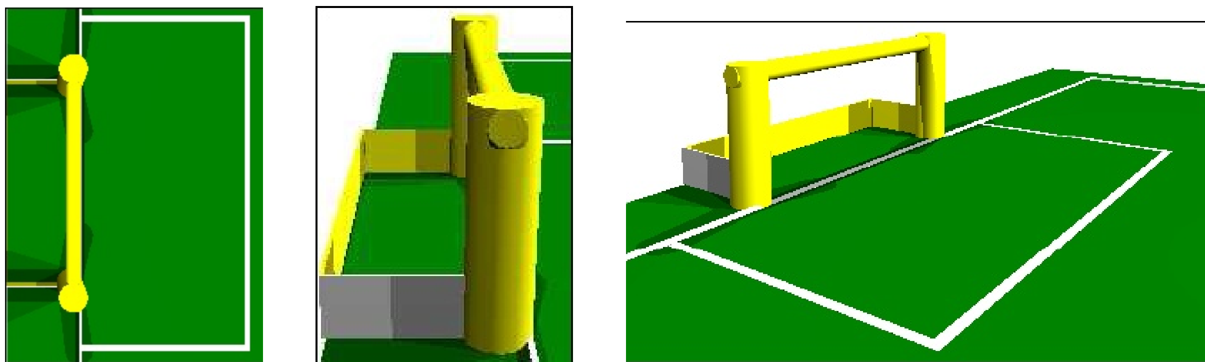


Figure 10.3: Above, side and diagonal views of the new goal.

a score. If a ball is kicked out of the field it is placed back at the center of the field. Since the 2006 penalty shootout rules are used, the attacking robot can not enter the yellow penalty area to shoot (the robot may enter the blue penalty area). If the ball goes inside the yellow penalty area but not into the goal, it will be replaced at the center of the field. Additionally if the attacking robot touches the ball inside the yellow penalty area the ball will be removed and replaced at the center of the field"
Figure 10.3 shows different views of the new goal. Detailed information about the rules and the goal dimensions can be found in [1].

10.2.2 Problem Analysis

One of the major problems is the free space in the back side of the new goal. The audience or the background detected in that free space could divert the robot. Dependent on the robot's perspective, the seen shape of the new goal changes much more than with the old one. Another problem is the very thin crossbar with a radius of only 25 mm. The bottom half of the crossbar always appears very dark; the round crossbar casts shadows on itself, because the lights are installed under the ceiling. Furthermore, there are always some highlights on the top half of the crossbar. So there is only a small, noisy strip of yellow color-classified pixels. All that makes detecting the crossbar very difficult. Additionally the crossbar disappears completely, if the robot is close to the goal. Unlike the old goal, the new one also has a very thick, round, and yellow colored goalposts. These must be considered in the calculation of the free part of the goal. Otherwise, the goalpost would be considered as a free part of the goal and the ball would just rebound when it is kicked against it.

10.2.3 Our Approach to the Challenge

We used the *GT2005GoalRecognizer* (see [15]) with some modifications for the new goal challenge. These modifications effect parts of the goal detection and the calculation of the free part of this goal.

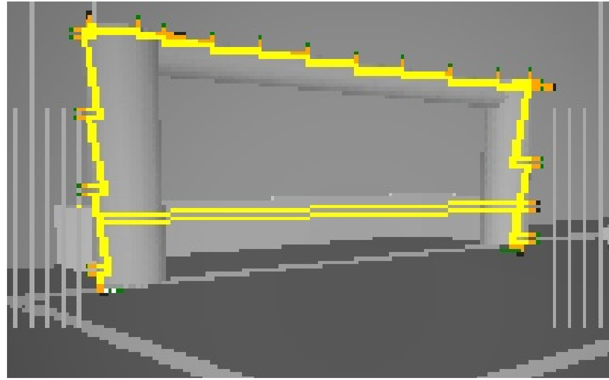


Figure 10.4: The zigzag scanline of the goal recognizer. Two different scanline types are used alternately: one scanline type is running along the outer border of the goal. The other scanline type is scanning towards the border to get in touch with it again. The lower parallel double scanline is checking the free part of goal.

10.2.3.1 Detecting the Goal

The goal recognizer used by the GermanTeam in 2005 does not consider the whole goal area, but only the border of the goal. The ImageProcessor searches in the main grid for a goal-colored blob (in case of the new goal for a yellow blob). From this position, the goal recognizer is triggered. First it tries to detect the left goalpost by scanning along the edge of this goalpost in a zigzag-line. This zigzag-line consists of two alternately used scanline types: one scanline type is running along the edge of the goal. To be sure not to cross the goal border with this scanline, it has a small inward rotation. The other scanline type is scanning towards the edge to get in touch with it. From the top point of the detected left goalpost, the crossbar detection starts. The zigzag-line follows the crossbar edge as far as possible to the right side. From that point, the right goalpost is scanned downwards in the same way like the left goalpost. Figure 10.4 illustrates the scanlines of the goal recognizer. Out of the collected information, a goal hypothesis is created as a bounding box around the detected goalposts and the crossbar. Dependent on the seen situation, multiple hypotheses can be found. Two further steps of the goal recognizer are merging of hypotheses and selecting one hypothesis which becomes the final goal percept.

Scanning in the described way does not consider the inner area of the goal. Thus, if both of the goalposts and the crossbar are well color-classified in the image, the detection of the new goal makes no problems even using the *GT2005GoalSpecialist*. But, as already mentioned, the crossbar is very thin and there is only a small, noisy stripe of yellow color-classified pixels.

Scanning along the crossbar is similar to scanning the goalpost; alternating, an inward orientated scanline along the edge of the crossbar and then a scanline to get in touch with the edge again, are generated as long as there are enough yellow classified pixels to assume that we are still within the goal. The scanline along the edge is generated by the method called *scanAlongLine()*, the scanline towards the edge by the method called *detectEdge()*.

The *scanAlongLine()* method generates a scanline that stops either after finding an edge or if it reaches the maximal length of twelve pixels. In the case of reaching the maximum length the

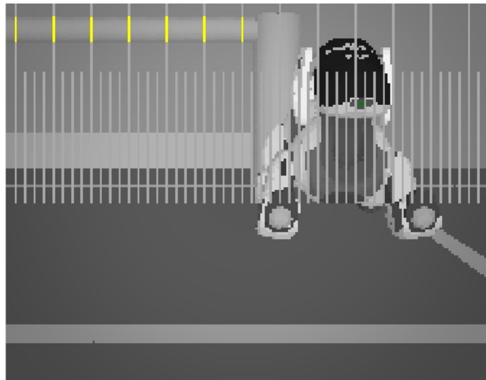


Figure 10.5: The longer vertical scanlines make it possible to find the goalpost (see yellow marked parts). The original scanlines are too short and would not discover the crossbar.

alternating routine continues, that means the method *detectEdge()* is called.
Two improvements were necessary for the *scanAlongLine()* method. First, the maximal lenght was changed to six instead of twelve pixels, because with regard to the thin crossbar it is recommend to make smaller steps. When *scanAlongLine()* detects an edge during scanning along the crossbar from the left to the right side, then we assume to have found the right goalpost. But because of noise and the slim crossbar, the end of the crossbar is often erroneously found too early. So the conditions for the decision if we either found an edge or not, have to be modified too.
For detecting an edge, two counters are used: the *edge-counter* and the *deviation-counter*. If a *gray* or *noColor* classified pixel was detected, then the color distance (in the color space) between the color of the detected pixel and the average color of all goal classified pixels is calculated and compared to a threshold. If the distance is greater than the threshold, we increase the *deviation-counter*. If a pixel of any other color class, except yellow, was detected, we increase the *edge-counter*. If one of these counters becomes greater than its threshold, the scanline assumes to have found an edge. See [15] for more details.
Doubling the threshold values makes the scanline more tolerant to noise. We also tolerate the appearance of a single non-yellow classified pixel. That means the counters can only be increased, if there are at least two of those other colored pixels on the scanline. These changes made it possible to travel from the left to the right goalpost successfully in many more cases.
The main scan-grid of the ImageProcessor was modified too. Originally, all vertical scanlines started fifteen pixels above the horizon. Now, every fourth vertical main-grid scanline starts at the top of the image. This is important for the case that the robot stands in front of the yellow goal, but only the crossbar and the right goalpost can be seen. The original scanlines are too short to find the crossbar. But the new longer scanlines discover the crossbar, and goal detection is triggered from this position. Figure 10.5 illustrates the described situation.

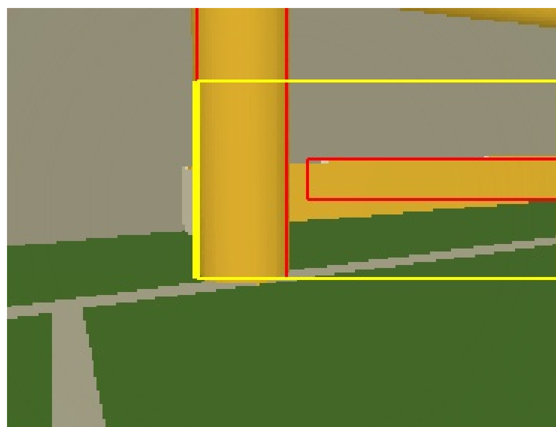
As already mentioned, the goal recognizer creates a goal hypothesis out of the found goalposts and the crossbar percept. In some cases multiple goal hypotheses are detected. After the complete main-grid is scanned, the next step is to merge goal hypotheses within the *mergeFragments()* method, if it can be assumed that two detected hypotheses belong to the same

object, and finally to decide which of the left hypotheses becomes the final goal percept. A lot of parameter fine tuning was done in the method *mergeFragments()*. Some of the changes will be described here.
The goal hypotheses (sometimes fragments of the goal), that were merged together in the original goal detector, had to be very similar in their size and their width-to-height ratio. Both hypotheses had to be located very close to each other in the image. These criterions were useful, because the old goal always appeared in the image as a block area. But the new goal appears in very different shapes depending on the robot's point of view. One problematic situation is shown in figure 10.6(a). Hier two hypotheses were detected: the left goalpost and the bottom back part of the goal. The original *mergeFragments()* method would not merge them and one of these hypotheses would become the final (and wrong) goal percept.
Since this is a frequent situation ,additional conditions are necessary to recognize the goal correctly in such a situation.

Lets call the left hypothesis in figure 10.6(a) *fragment i* (the left red box around the goalpost) and the right *fragment j* (the right red box around the bottom part of the goal). Both fragments in this figure will be merged, if the following conditions are complied:

- the area of both fragments should be large enough (more than 100 pixels, assuming that the robot is close to the goal)
- the height of fragment i should be greater than its width
- the width of fragment j should be greater than its height
- the range of the y-coordinates of fragment j should lie completely within the range of the y-coordinates of fragment i
- the area of fragment i should be smaller than six times the area of fragment j
- the area of fragment j should be smaller than six times the area of fragment i

After finishing all possible mergings of goal hypotheses, the method *interpretResults()* is called. It chooses one of the remaining goal hypotheses and creates the final goal percept out of it. Here it was necessary to relax some conditions. Figure 10.6(b) shows a situation, where the robot is standing in front of the penalty area looking straight ahead towards the goal. No goalpost and no crossbar can be seen in the image, only the bottom part of the goal. The original *interpretResults()* method only considers a goal hypothesis with a maximal width of five times the height of the hypothesis. This condition was relaxed by doubling this ratio and therefore hypotheses with a very large width (in some cases the whole image width) and a reasonable height are also candidates to become a goal percept.



(a) The red boxes indicate the goal hypotheses. The yellow box indicates the final goal percept after merging the two hypotheses.

(b) Robot is standing in front of the penalty area. A very wide goal percept was detected.

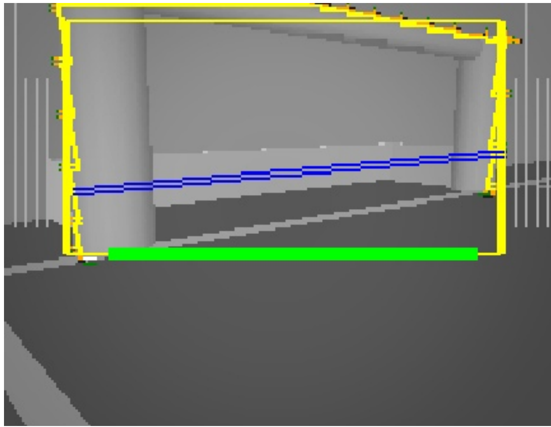
Figure 10.6: Goal hypotheses (red boxes) and the final goal percept (yellow box) in different typical situations.

10.2.3.2 Detecting the Free Part of Goal

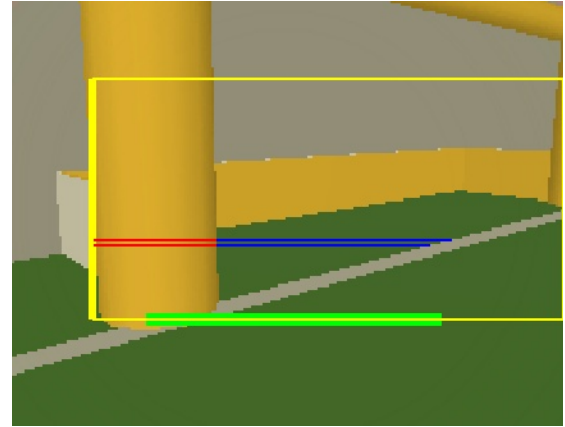
One part of the percept is the information about the free part of goal. It is important for shooting the ball into the goal and to avoid the goalie. A parallel double scanline checks the pixels between the detected goalposts (see the lower parallel double scanline in figure 10.4).

Usually this double scanline is running from one goalpost to the other. But it is possible, that one of the goalposts is covered, for instance, by a robot or is not completely within the image anymore. If in that special case the height of one goalpost is three times greater than the height of the other, the scanline starts from the greater goalpost and runs parallel to the horizontal image border. The scanline searches for edges in the same way it happens during crossbar detection. So the yellow classified space between a goalpost and a found edge is assumed to be the free part of goal.
The height of the starting point of the double scanline relative to the gound had to be adapted for the new goal, too, because otherwise the scanline would run over the aperture and not along the yellow bottom part of the goal. Thus the starting point has been lowered (see figure 10.7(a)). While the double scanline scans within the bounding box of the final goal percept, it now accepts green classified pixels as free part of goal pixels. This is important, if the robot is looking at the new goal from the side. Otherwise, the free part of goal would be calculated wrongly (see figure 10.7(b)).

To avoid that the ball rebounds from the large goalposts, their width is considered in the calculation of the free part of goal by cutting away the outer part of the goalpost. Only the inner part of the goalpost is considered to belong to the free part of goal, because when the ball hits the inner part of the goalpost, it is very likely that the ball rolls into the goal anyway. Relative to the height of the detected goalpost, the amount of pixels that will be cut away from the free part



(a) The blue double scanline checks the free part of goal. The green stripe shows the resulting percept of the free part of goal. The widths of the goalposts are considered in the calculation.



(b) Without accepting green pixels along the scanline, the free part of goal would end in the right side of the left goalpost (red scanline part). The modification of the scanline allows detecting the free part of goal more accurately.

Figure 10.7: The double scanline is searching the free parts of goal. The free part of goal percept is indicated by the green stripe. The outer parts of the goalposts are not included. The yellow bouding box shows the resulting goal percept.

of goal percept is calculated. The calculation is very easy: The goalpost has a height of 350 mm and we want to cut half of the goalpost's width, ergo 50 mm.

$$pixelsToCut = \frac{50 * goalpostHeight}{350}$$

Depending on the point of view, the goalposts can have different heights in the image. Thus the amount of pixels to be cut away from the free part of goal is calculated independently for each goalpost. The result is shown in figure 10.7. The green stripe shows the free part of goal percept. The outer parts of the goalposts are not included.

10.2.3.3 Behavior

After booting, the robot starts searching for landmarks immediately in order to get a good self localization as early as possible. To bring the ball near to the goal, the first kick is always a soft chest kick. To perform a hard kick directly would be too risky. The ball might stop in the penalty area when the kick is not executed perfectly. Then the robot tries to grab the ball and score. He rotates with the grabbed ball into the direction of the goal. If the goal is not seen in this state for 2500 ms, a soft kick is performed again. Otherwise, if the angle to the free part of goal is between -10 and +10 degrees, a hard kick is performed. If the ball cannot be seen for a few seconds, a searching routine tries to find the ball again.
Every time after a goal was scored the referee presses the back button and moves the ball into the center circle (see New Goal Challenge rules). If the back button was pressed, the robot

immediately starts to walk to the middle circle until it is able to see the ball again. Then the robot grabs the ball, rotates into goal direction and starts the complete behavior routine again by bringing the ball nearer to the goal with a soft chest kick.

A special feature that is used during ball searching, approaching and handling, is the collision detection. This is very useful, because the positions of the opponent robots are unknown before the competition, and getting stuck with other robots can waste a lot of the limited time. If a collision is detected, the robot moves backwards very fast for about one second. Simultaneously, it moves into the opposite direction of the detected collision side.

10.2.4 Evaluation of the Results

As illustrated in figure 10.6, the bounding box of the goal percept in the image does not always contain the whole goal (in 10.6(a) and 10.6(b), the yellow box should touch the top image border). This could lead to some problems concerning the self locator because of possible wrong distance calculations. If the robot is close to the new goal, the architecture of the new goal can potentially give a lot of additional information which could be used to determine the distance more accurately (for instance the seen width of the goalpost or the height of the bottom part of the goal). This information could be used for generating the goal percept as well. But there was not enough time left to integrate these features. For this challenge, it was more important to get accurate information about the free part of goal. In almost all situations, the free part of goal percept was very useful. The behavior is quite simple and tends to shoot at the goal very often. This seems to be naive at first view, but if the ball misses the goal and leaves the field, then the referee replaces the ball manually into the middle circle. It is much better to handle the ball starting from that point than from anywhere near the field corner (and thus in a disadvantaged angle to the goal).

In the competition, the opponent players were not distributed along the whole field, but they were all standing between the goal and the middle circle. This was the worst case situation for our strategy. We were able to achieve only one goal and so we shared the second place with four other teams. We had another very good goal chance, but we could not seize it: three times in a row the robot tried to execute the hard kick just in front of the goal close to the penalty area, but the kick execution was always disturbed by getting stuck in one of the opponent players' legs with the left paw. After the third try, the ball just rolled into the penalty area and had to be removed by the referee. Collision detection was not active during kick execution or other special actions.

10.3 Open Challenge

The Open Challenge gives teams a chance to present the results of their research in a way they can choose themselves.

10.3.1 Challenge Rules

There are hardly any rules for the open challenge. Teams are given 3 minutes to perform or present a feat to the other teams (and the audience). This can be related to the game, but it can also be any other performance involving the robots, or even simply a presentation of a particular scientific or engineering result made by the team. Every team gets to rank the other teams as it sees fit, resulting in an overall winner.

10.3.2 Our Idea

We wanted to present our results in modelling other players on the field, particularly the co-operative modelling using the wireless communication. To that end, we put a number of robots on the field: One of them, wearing a red jersey, is in effect blindfolded, i. e. all optical processing is turned off, the other or others are meant to detect him, share the percepts, resulting in a collaborative model of this player. Using this information, the blind player is meant to perform as if it was able to see, and walk to the center circle and perform a special action to prove the accuracy of the model.

10.3.3 Blind Dog

10.3.3.1 Player Model

The core of the player model was the model described in chapter 5. However, since the accuracy and validity of the model was crucial, further post-processing was applied to the general purpose model. Most of this post-processing is a result of an extra piece of information the challenge player model has: there is only one player on the field. The challenge player model was designed to err on the side of caution: rather than giving out information that seems uncertain or unlikely, no position is returned. This is due to the fact that a wrong localization can have catastrophic results: the blind player moves towards the wrong direction, and the observing players lose sight of him. This is amplified by the difficulties of detecting players at medium to high distances.

A huge problem was that on the competition site, detecting players was much harder since the colors of the landmarks were not made to specification. Both the sky blue and the pink was closer to the player colors than it should have been, making the creation of a color table that could both be used in localization and for player detection difficult at best. Of course, localization and accurate player detection are both central to the working of the challenge.

Finally, the player detection does not contain any reliable information on the orientation on the field. This is not a big issue in opponent player modelling, but absolutely crucial for the blind player. An approach to compute the orientation based on the deviance of the observed from the modelled position was implemented. Unfortunately, the detection and modelling are not accurate enough for this approach to work, so the blind player had to be started with a known orientation, with the future orientation based solely on odometry.

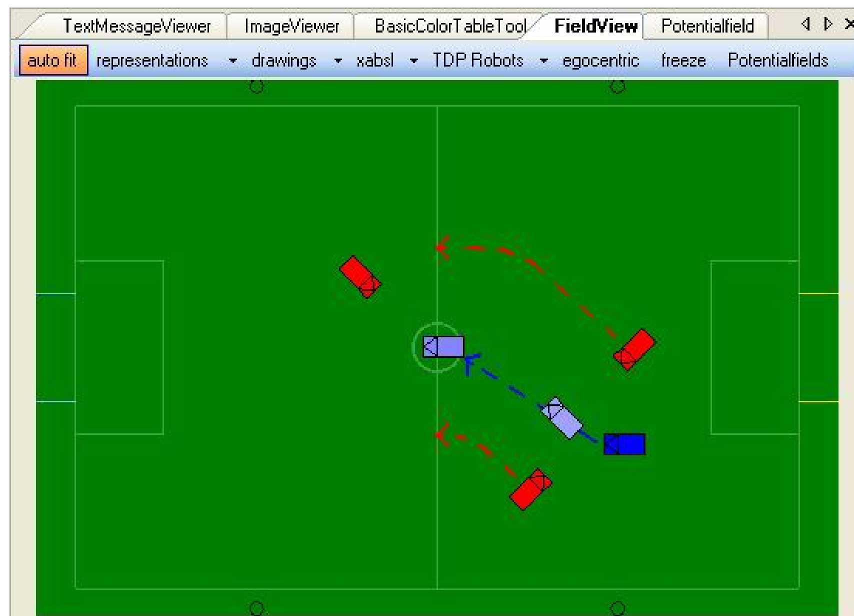


Figure 10.8: The two observers (red) lead the blind blue robot to its destination in the center circle, where a third observing robot is already waiting to confirm the arrival.

10.3.3.2 Different Behaviors

Two observers, one blind dog and one finisher. In this constellation we have two robots (called "Buddy1" and "Buddy2"), to lead the blind one (called "Player") to its destination, and an additional robot ("Target") to confirm the approach of the destination.

At first all three red robots have to localize themselves and search for "Player". If a blue robot is seen, the position is communicated via wireless network to him. "Player" has to model its own position depending on the percepts it receives from the others and move to its destination, if the validity is high enough. For this, one "Buddy" has to stand and observe "Player", while the other one walks well localized to the next observing point, where it will stop and lead "Player" further ahead. The third robot, Target, walks after selflocalization and "Player" is localized, to a position behind target point and waits for Player.

Two observers and one blind dog. Another approach is without "Target", because most of the time he would wait for "Player", who has to be localized all the time. So it's not necessary to have the position validated by three robots. With only two observers the relative starting position of both red players to the blue one must be defined more accurately, because two dogs can't search the whole field accurate enough or would need too much time.

One observer and one blind dog. With two observers it is necessary to model the observers and to walk to specified positions, one on each side of "Player", in order not to collide and delocalize each other. To prevend this, only one observer could be used (see Figure 10.9). This

Figure 10.9: A simpler approach uses just one observer which tracks the blind robots position.

robot should always look at “Player” and should be well localized, because “Player” only gets percepts from this robot and has to trust most of them.

10.3.4 Evaluation of the Results

When we gave the presentation, the blind player would only get to the center about a third of the times. The typical cause of failure was a wrong modelled position of the player, with it wandering off out of the visual range of the observer, making a redetection highly unlikely. The inaccuracies of the model were usually due to either a wrong localization of the observer or a misdetection of the player, both due to the difficulties of the color table. However, when we performed the feat, it worked perfectly on the first try. The other teams seemed to be impressed without it, though, ranking us at third place in the Open Challenge.

Chapter 11

Competitions

This year we took part in three major competitions. One was the RoboLudens in Eindhoven (Netherlands), the other one was the US Open in Atlanta (Georgia, USA), and the most impoertant one was the RoboCup in Bremen (Germany).

11.1 RoboLudens

This year, the European Championship in Robot Soccer was held in Eindhoven in the Netherlands. It was called RoboLudens and took place from April 7th to April 9th 2006.
The Microsoft Hellhounds scored very well at this competition. After a daunting start, we continually improved our performance and eventually became European Champions, thus vindicating the title from the GermanOpen in Paderborn 2005.
In the preliminaries, we had to accept two defeats, first against the Darmstadt Dribbling Dackels and later against the Aibo Team Humboldt, and also a draw against the Dutch Aibo Team. But we accomplished to beat three other teams with ease: the Hamburg Dog Bots, Team Chaos, and the Bremen Byters. After these six preliminary Games we finished ranking third (behind the Darmstadt Dribbling Dackels and the Dutch Aibo Team) with ten points and a goal ratio of 18:6.
We managed to fix a bug in the behavior of the striker in front of the opponent goal right after the preliminaries, but in the quarterfinals, the wireless network was causing quite some trouble. Most likely due to that, we outplayed the Hamburg Dog Bots by a narrower margin than the first time and even had to concede a goal against.
After solving some of the problems with the crowded access points, our team communication worked somewhat better. This way we could prevail over the Dutch Aibo Team in a very thrilling semifinal game, scoring 2:0.
Our opponent for the final was the dreaded Aibo Team Humboldt, but we defeated them by 4:0 and became European Champion.

As usual, the teams were also invited to attend the Technical Challenges. The Microsoft Hellhounds performed very well in the New Goal Challenge and the Open Challenge, but did not compete in the Passing Challenge. Accordingly, we rounded off third with 45 Points, right

PRELIMINARIES	
Microsoft Hellhounds - Dutch Aibo Team	**1**:1
Hamburg Dog Bots - **Microsoft Hellhounds**	0:**4**
Darmstadt Dribbling Dackels - **Microsoft Hellhounds**	3:**0**
Microsoft Hellhounds - Aibo Team Humboldt	**0**:1
Microsoft Hellhounds - Team Chaos	**6**:1
Bremen Byters - **Microsoft Hellhounds**	0:**7**
QUARTERFINALS	
Microsoft Hellhounds - Hamburg Dog Bots	**3**:1
SEMIFINALS	
Dutch Aibo Team - **Microsoft Hellhounds**	0:**2**
FINAL	
Aibo Team Humboldt - **Microsoft Hellhounds**	0:**4**

Table 11.1: The results of the Microsoft Hellhounds at the RoboLudens in Eindhoven 2006

after the Darmstadt Dribbling Dackels and the Bremen Byters.

11.2 US Open 2006

The US Open were held in Atlanta this year. This competition was especially interesting because it was the first time this year that we had the opportunity to play against the American teams, who would also compete against us in Bremen. Aside from that, one long awaited match would be that against our favorite opponent, the team from CMU.
During the first tests, it was realized, that the conditions at the competition site weren't very good: The lighting conditions were suboptimal, and also the carpet wasn't what we were used to. Surprisingly our kicks and walks still worked, while the grabbing didn't perform well. Nevertheless we successfully contested in our fist games: The fist game versus Northern Bites could be decided 6:0 for us, and the game Microsoft Hellhounds vs. Demonstra Dogs ended 8:0. A 6:0 success versus Austin Villa the next day secured our move-in to the semi finals. During the semi-final-game against UPenn we received the first goal against us. No need to worry, though, since we shot four goals ourselves. This meant, that we would play in the final, and our opponent for this match just happened to be the team from CMU! This exciting game then ended 5:2 in favor of the Microsoft Hellhounds, and we were the winner of the US Open 2006.

11.3 RoboCup 2006

Nearly at the same time as the human soccer world championship in Germany, the world championship in robot soccer began in Bremen on 14th of june. It lasted until the 20th of june and

ended with a satisfactory third place in robot soccer and a first place in the technical challenge for the Microsoft Hellhounds team. After the great success of winning the world championship in the last year as part of the GermanTeam, this year the Microsoft Hellhounds team proved itself as a serious competitor. Because of this, the duel GermanTeam vs. Microsoft Hellhounds was expected as one of the top matches. The strength of the Microsoft Hellhounds team was great enough to win all of the matches in the first, the intermediate and the second round. In average, the team scored seven goals in the first seven games against opponent teams like SPQR, Upenn or Northern Bytes. The only goal an opponent team scored against the Microsoft Hellhounds was shot in the match against Cerberus from Turkey, which ended 6:1. The quarter final against Wright Eagle was a very hard match. Due to problems with the network, the Microsoft Hellhounds team scored once in the second half and the match finished 1:0. The semi final was reached. The NUbots from Australia, the opponent team in the semi final, was an outstanding team with great moves and teamplay. In the end the robots from the Microsoft Hellhounds had no chance to score and lost the game with 0:7. This was hard to believe, but indeed this result was until then the worst result the NUbots achieved during the competitions. In the end they won the RoboCup 2006 against the other australian team rUNSWift with 7:3. The other team from germany lost the semi final as well and the GermanTeam was our next opponent in the match for the third place. In the end, it was a clear victory against the GermanTeam with 0:6 goals. The last thing to tell about the success of the Microsoft Hellhounds team on the RoboCup 2006 is the first place in the technical challenge. The second place in the Passing- and third place in the New Goal- and Open challenge secured our victory in this discpiline. In the end, the Microsoft Hellhounds achieved a great result at the RoboCup 2006.

Chapter 12

Conclusions

This year's project group achieved almost all of their individual and team objectives, and the results turned out well. The group grew together quickly and joined willingly in some "extra-curricular" activities like the attendance of the Imagine Cup and some festive barbecues.
Consequently, the collaboration was quite fruitful, and the team was also very successful at all competitions that were attended.
Below, the conclusions of the specific working areas are recapitulated at a glance.

12.1 ImageProcessor

The usage of two color tables brings more stability into the ball recognition, especially when the robot is close to the ball. It also relaxes the trade-off during the color table creation process and therefore it saves time, because neither of the color tables needs to be as universally applicable as before with only one color table.
The calibrated motion compensation was tested in more than 5000 measurements on different robots, whereas the standard deviation of the measured head rotation angle went down to less than half of the old divergence. Therefore, the percepts calculated during fast head movements are now more accurate than before.

12.2 Horizon Calibration

The horizon-based calibration of the camera matrix was a success. Even though the automatic detection did not work due to the low resolution of the images, the calculations done by the evolutionary algorithm could still be used. The horizon had to be found manually, but the results were very satisfactory. The position of the horizon could be corrected for every head position. Admittedly, some problems could not be fixed so far, but those would probably involve a calibration, based not only on the current head position, but also on head movement.

12.3 Ball Locator

The first versions of the new Kalman BallLocator indicated the position and velocity modeling better than the GT2004BallLocator in about 60% of all frames even with the old ‘intuitive’ methods of measurement covariance matrix adaption still used.
After introducing additional improvements, the results were 60-70% better in position and about 50% better in velocity.
The latest Kalman BallLocator reacts on a shot almost instantly. It has a high accuracy in position; aeven when the ball is moving, its velocity is far less jumpy, and its direction is very close to the real movement direction. A modelled static ball does not move significantly even when the robot’s head is heavily panning, and the ball in the model ‘rolls’ out very close to its real position when it cannot be seen any more. When the robot is turning around searching for the ball, the odometry adaption finally works now, and the modelled ball stays at its position relative to the robot. The new ball locator can even track a ball when the robot is running behind several other robots which are obstructing the vision.

12.4 Opponent Player Modelling

Under good conditions, the opponent player model provides sufficient results. Cooperative modelling, with percepts shared between robots, can further improve the accuracy. However, further work needs to be done in that area, creating as robust a model as possible under the circumstances. A better players detection would make creating that model much more easily achievable. On the other hand, any model is only as useful as it is being made use of. Feedback from engineering behaviors can be used to improve and adapt the model to fit more specific needs, as was seen in the RoboCup Open Challenge. This is especially true for a model where, due to technical limitations, certain trade-offs are unavoidable.

12.5 Behavior Learning

The evolution of behaviors proved to be very useful for training complex skills that are difficult to fine-tune manually. The ball-acquisition could be improved significantly by the learning process that was implemented. Both strategies, the (1+1)-evolution algorithm and the Experimental Search Space Exploration, evinced advantages and drawbacks as opposed to each other, and both lead to virtually optimal values for the “grab-ball” behavior.
Hence, an adequate way to make behaviors easily adaptable to changed circumstances was created. For competitions, the optimal values can be predetermined in the lab, and the last adjustments can then be obtained quickly at the venue.
These prospects encourage to continue the work on learning methods for behavioral tasks and to identify more areas where they could be utilized.

12.6 WalkingEngine

The various efforts trying to correct the accuracy of the walks showed the complexity and diversity of this problem. The method of correcting *WalkReqest*s while continuosly taking measurements was capable of reaching a high precision, but it was also limited to correct just a small spectrum of *WalkReqest*s. As accurate as the previously measured odometry itself is the method of inverting the odometry table.
A lot of time of the optimization has been spent on fine tuning the *AIBO*s' walking behavior. The correction of polygon sets proved to be the most effective procedure, resulting in optimized gaits.

12.7 Special Actions

Many new and useful special actions were created and old ones were revised conveniently with the help of the new motion designer tool. A list of the currently available kicks is provided in the appendix.

12.8 Tools

Some additional Tools were developed last year and also already used.

12.8.1 RobotControl XP

This program was developed anew, because *RobotControl* and *RobotControl* 2 had some stability and functional disadvantages. The result is a more stable and faster tool. Some functionalities are still missing, but can be ported from *RobotControl* 2 in some short steps.

12.8.2 Potential fields

The visualization of potential fields was useful to create new fields and test them. It could be extended by dynamic resolution and area selection in *RobotControl*, so that it would also be possible to zoom in and get a higher resolution.

12.8.3 BasicColorTableTool

Some new functions and hot key functions of the *BasicColorTableTool* offer a way to design color tables more quickly and comfortably. The tool facilitates the dealing with the new *colorTable128* format and the associated *rangeThreshold* value. Of course, it can still handle the old *colorTable128* format.

12.8.4 HorizonCalibrationTool

As already mentioned, the camera matrix, which is calculated in each image, needs to be corrected. This user interface is a good way for doing this (cf. Section 12.2).

12.8.5 ImageViewer

The ImageViewer already existed, only some functionality was missing. Now it is possible to calibrate the channel thresholds and to visualize their corresponding gradients.

12.8.6 Motion Designer

The MotionDesigner is a huge and powerful tool for creating special actions. It makes creation of special actions easier and faster, and is a great improvement to the old motion designer in *RobotControl*.

12.9 Challenges

We participated in all three challenges and won the overall first place at the world championship in Bremen. Here is a short description of them.

12.9.1 Passing Challenge

If the cirsumstances are favorable and the behavior is well-adapted to the prevailing conditions, then the passing challenge works very well: the robots are able to pass the ball around effectively and without return passes.

12.9.2 New Goal Challenge

With some modifications, the new goal could be identified quite reliably by the goal recognizer, and the “free part of goal” percepts were very useful. Thus, we were able to achieve one goal and shared the second place with four other teams.

12.9.3 Open Challenge

The idea of a blind robot was converted well at the RoboCup 2006 and the demonstration worked perfectly. Although the colors were bad and hard to distinguish in the colortable, the blind robot was recognized and enabled to move to the center by the accompanying robot.

12.10 Competitions

Altogether, the Microsoft Hellhounds had a very successful year again. We achieved three first places, one for the soccer competition in Eindhoven, one for the soccer competition in Atlanta, and one for the technical challenges in Bremen. Thus, we became European and US champion. Additionally, we rounded up third in the soccer competition at the world championship in Bremen and also in the technical challenges in Eindhoven.

Appendix A

PotentialFields

This text is written for some help with potential fields and describes the structure of the ẊML-files.

A.1 Hints for RC2

At the moment the makefile at "$GT2005\Src\Modules\BehaviorControl\-MSH2006BehaviorControl\PotentialFields" is executed by the send-button, which compiles all potential fields in this directory. After this the .pfc file under "Pfield\MSH2006" (in config-directory) is send to the robot.

A.2 Comments

First it could be useful to comment your files:

```
<!-- This is a comment -->
```

A.3 Header

All potential fields starts with a similar header, so copy these lines and use it.

```
<!DOCTYPE potentialfields-configuration SYSTEM
    "../../../../Tools/PotentialFields/pfc/pfc.dtd" [
<!ENTITY standard-object-state-symbols SYSTEM
  "./common/standard-object-state-symbols.xml">
<!ENTITY standard-obstacle-objects SYSTEM
  "./common/standard-obstacle-objects.xml">
<!ENTITY standard-obstacle-instances SYSTEM
  "./common/standard-obstacle-instances.xml">
]>
```

```
<?xml-stylesheet type="text/xsl"
  href="../../../../Tools/PotentialFields/pfc/transform-pfc.xsl" ?>
```

This includes the Document-Type-Definition for PotentialFields, some standard symbols and the conversion file for robot compatible pfc-files. The entities are optional and could differ between files.

A.4 PotentialField

The potential field is defined by an element called potentialfields-configuration and identified by a name. This is only used once per potential field.

```
<potentialfields-configuration name="One potential field"
    description="you never walk alone">
<!-- define field here-->
<\potentialfields-configuration>
```

A.5 Structure of potentialfields-configuration

This Element is structured by three defined sections, which have to be present:

```
&standard-obstacle-objects;
&standard-object-state-symbols;
&standard-obstacle-instances;
```

A.5.1 Standard-obstacle-objects

In this Element all Objects, which will be used later, are defined. It's like a class or a template.

```
&standard-obstacle-objects;
```

Behind this line one Template for each class of objects is created, which describes the object-properties like

- function: no-function, linear-function, parabolic-function, asymptotic-function, social-function
- appearance: point-field, shape-field, sector-field
- geometry: no-geometry, line, polygon, circle, pt

The standard objects are called `object` and mostly used. The parameter **"type"** is required and possible types are "attractive", for pulling the robot, and "repulsive" for pushing it away. You can optionally define the field as a **"tangential-field"** with direction "clockwise", "counter-clockwise" or "none" (`tangential-field="clockwise"`), and give a **"description"**.

```
<object name="Destination-Wide" type="attractive"
    tangential-field="clockwise">
<!--function-->
<!--appearance-->
<!--geometry-->
</object>
```

A.5.1.1 Functions

Each Function describes how the influence of this object decreases with growing distance.

- no-function: Objects without any influence
- linear-function (see Figure A.1): $f(x) = a * x + b$
  $a = -atZero/range$;
  $b = atZero$;

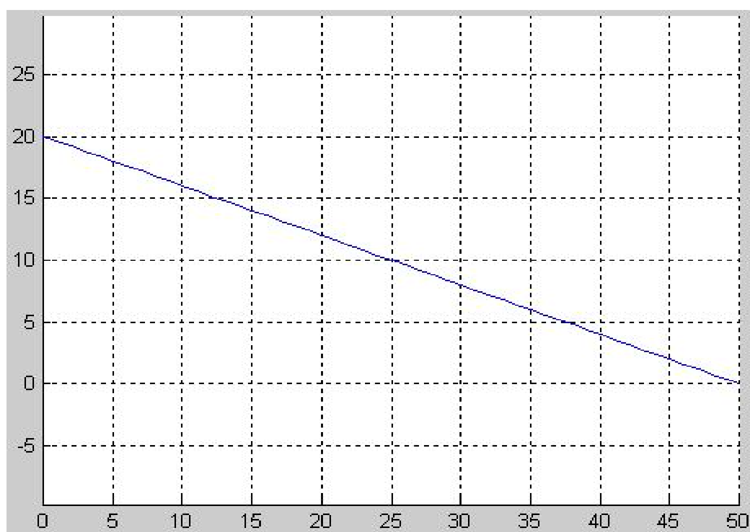


Figure A.1: Linear Function: $range = 50$, $atZero = 20$, x is the distance and $f(x)$ the influence on the field

- parabolic-function (see Figure A.2): $f(x) = a * x^2 + b$
  $a = -atZero/(range * range)$;
  $b = atZero$;

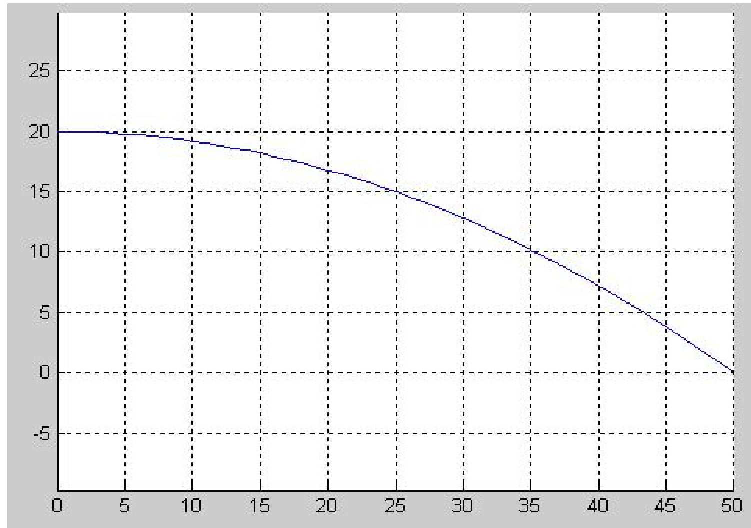


Figure A.2: Parabolic Function: $range = 50$, $atZero = 20$, x is the distance and $f(x)$ the influence on the field

- asymptotic-function (see Figure A.3): $f(x) = a/x + b$
  $a = atZero/(1.0/solidCenter - 1.0/range)$;
  $b = -a/range$;

- social-function: $s(x) = -\frac{c_1}{x^{\sigma_1}} + \frac{c_2}{x^{\sigma_2}}$ with optimal distance: $d = (c_1/c_2)^{\frac{1}{\sigma_1 - \sigma_2}}$

  Parameters: repulsive-constant (c_1), repulsive-exponent (σ_1), attractive-constant (c_2), attractive-exponent (σ_2), const-interval (below this value the function is constant), k (optional, const additional factor).

  This is a special function with an attractive and a repulsive part and without maximum range. It's used for formation between robots. For additional information read the diploma-thesis from Tim Laue[11].

A.5.1.2 Appearances

For each object the source of the field must be set. These geometric alignments can be chosen:

- point-field: The resulting field is centered in this figure

- shape-field: The influence is away from the shape of the figure

- sector-field: This field is inside the given opening angle with a maximum degree of the half of the opening-angle to the left and right. The field itself is defined by the given function, but its effect to the side, away from zero degree, is defined by the cross-function.
  Parameters: opening-angle, cross-function.
  Cross-function can be one of these functions: parabolic-function, linear-function, asymptotic-function.

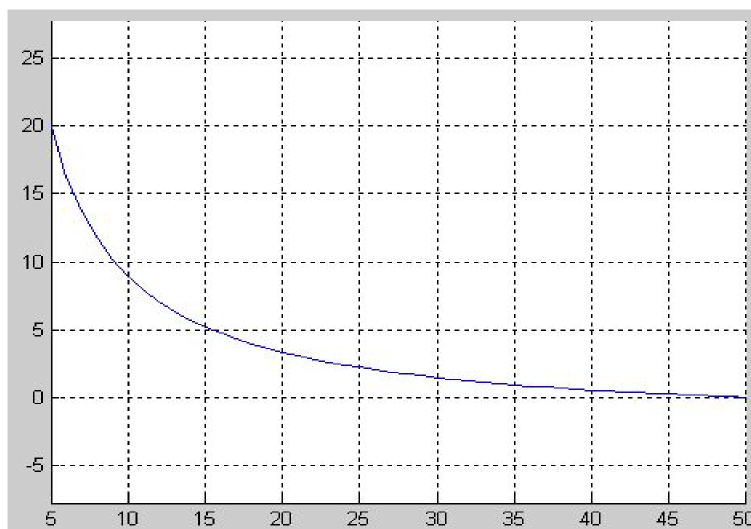


Figure A.3: Asymptotic Function: $range = 50$, $atZero = 20$, $solidCenter = 5$, x is the distance and $f(x)$ the influence on the field, for $distance < 5$ the influence is constant $atZero$

A.5.1.3 Geometry

These elements are available:

- no-geometry: without Parameters
- line: consist of two points (pt)
- polygon: described by at least three points (pt)
- circle: Parameters: radius, intersectable
- pt: Parameters: x,y

A.5.2 Object state symbols

This section is for a dynamic object-state. More Information can be read at [11].

```
&standard-object-state-symbols;
```

A.5.3 Instances

Single Instances. All objects must be instantiated. This section starts with `&standard-obstacle-instances;` and all objects are defined like this:

```
<object-instance type="Destination-Wide"
                                  name="destination-wide-radius">
<dynamic-pose get-data-from="destination1-symbol"/>
</object-instance>
<object-instance type="Own-Penalty-Area" name="own-penalty-area">
<static-pose x="-2050" y="0" rotation="0.0"/>
</object-instance>
<instance-group name="both" description="ups">
  <include name="own-penalty-area">
  <include name="destination-wide-radius">
</instance-group>
```

It is also possible to define formations, which are fields relative to objects.

A.5.3.1 Formations

These are relative objects which are adjusted to other objects and must be defined after object-instances. They could be used to create special formations between robots.

```
<formation-object name="in-front-of-ball">
<relative-to object="obstacle-ball" angle="45">
<linear-function at-zero="100" range="200"/>
</relative-to>
</formation-object>
```

A.5.4 Behavior

For the Behavior all objects must be combined. Therefore the Element `<potentialfield-composition>` is used. Within this area the `<motionfield name="go-to-pose-with-rotation">` defines one possible motion field.

A.5.4.1 Motionfield

Motion fields are used for robot movement and have some parameters.

Return Value. For the motion field `<return-gradient/>` or `<return-const value="42"/>` must be defined. These Flags set the validity of the action chosen by this motion field to the length of the gradient or to a fixed value. This is important if more than one motion field is defined, because the motion with the highest value will be executed.

including and combining motion fields. Some motion fields can be combined with other motion fields `<combine-with name="another-motionfield"/>`. In this case the results of both are added.

For objects the command is called `<include name="one-Object"/>`, for object groups `<include-group name="both"/>` and for formations `<include-formation name="in-front-of-ball">`. This will be used mostly in motion fields, cause they include the previously defined objects.

Other optional options.

- avoid-local-minima
- include-random-motion-generator

A.6 Additional features

- Behavior selection with an action field. This kind of action selection is very hard and much more difficult to write and debug than it is in xtc with kickselectiontable. At the moment it's never used, so that examples don't exist. For additional information read [11].

Appendix B

KickList

B.1 Kicks from grab

bashFromGrab. seems to be a front kick
didn´t worked on the carpet in the lab!

diveFromGrab. front kick, but it is very poor on the carpet in the lab

divekick. very soft front kick

fastExecutedKickEnhanced. middle range front kick (push), enhanced version of fastExecutedKickFinal

fastExecutedKickFinal. soft front kick (push), used during the Robo Ludens 2006
very straight, fast executed and worked very well!

fastExecutedKickFinalRobust. mod of fastExecutedKickFinal
not very reliably on the carpet in the lab!

frontKick1. middle range front kick (push)

frontKick2. another middle range front kick (push)

grabPressurekickMiddle. middle range and straight front kick
the robot press the head on the ball and pushs the ball away with the legs

grabPushFastExecuted. a variant of grabPressurekickMiddle

headKickFromGrab. middle range front kick
the robot lies down and kick the ball away with the head

headTapUNSW. between middle range and strong front kick
the robot lies down and kicks the ball with the head and the weight of its body

kickAlongArmLeft. very strong front kick
the robot kicks the ball with the chest and the head along the left arm

kickAlongArmRight. very strong front kick
the robot kicks the ball with the chest and the head along the right arm

nuFwdLeft, nuFwdRight. very strong forward head kick used during the RoboCup 2006
at the moment the kicks for the slow and the fast carpet are the same

B.2 Kicks not from grab

anyLeft, anyRight. middle range head kick to the left side or rather to the right side.

anyLeftBT, anyRightBT. a modified version of anyLeft and anyRight developed for the Robo Ludens 2006

armLeft, armRight. probably a kick to the left and the right side with the front leg

backKickLeft, backKickRight. kick to the left and right backside
didn´t worked on the carpet in the lab!

bash. seems to be a front kick
didn´t worked on the carpet in the lab!

bbHeadLeftSoft, bbHeadRightSoft. middle range head kick to the left and the right side
(about 90 degree)

bbHeadLeftStrong, bbHeadRightStrong. stronger variant of the previously head kick
(about 90 degree)

bubuUNSW. strong front kick

chestHard. rather a soft front chest kick

chestSoft. soft front chest kick
seems to be defective

executeForwardKick. strong front kick if it works, but a neck joint killer if not

forwardKickFast. middle range front kick

forwardKickHard. stronger variant of forwardKickFast

fwdHardUNSW. middle range front kick, not very sparing for the neck joint

headKickLeft45, headKickRight45. middle range head kick to a direction of 40-60 degree
the robot uses the legs to correct the direction

headLeft, headRight. very strong 45 degree side kick

headLeftSoft, headRightSoft. soft 45 degree side kick

hookLeft. strong 45 degree paw kick
the robot uses the right leg to kick the ball in about 45 degree to the left side

hookRight. strong 45 degree paw kick
the robot uses the left leg to kick the ball in about 45 degree to the right side

pushSoft. soft forward head kick

pushStrong. relative soft forward head kick

leftKick120, rightKick120. soft side kick in a direction of 120 degrees when the ball is in front of the robot

leftPaw, rightPaw. very soft forward kick
the robot kicks the ball mainly with the chest instead with the paw

slapLeft, slapRight. middle range backward kick
this kick works only when the ball is positioned next to the robot

References

[1] Committee Robo Cup 2006. Technical challenges for the robocup 2006 legged league competition, 2006. http://www.tzi.de/4legged/pub/Website/Downloads/Challenges2006.pdf.

[2] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[3] M. Broszeit, S. Czarnetzki, T. Diekmann, D. Fisseler, T. Kerkhof, M. Meyer, B. Schmitz, C. Rohde, X. Tran, T. Wegner, J. Winter, and C. Zarges. Pg468 report. Technical report, University Dortmund, 2005.

[4] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo Methods in Practice*. Springer, NY, 2001. Online: http://www-sigproc. eng.cam.ac.uk/~ad2/book.html.

[5] A. Ferrein, L. Hermanns, and G. Lakemeyer. Comparing sensor fusion techniques for ball position estimation. Technical report, RWTH Aachen, 2005.

[6] Peggy Fidelman and Peter Stone. Layered learning on a physical robot. In *Under Review.*, 2005.

[7] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice Hall, 1988.

[8] C. C. T. Kwok, D. Fox, and M. Meila. Real-time particle filters. In S. Becker, S. Thrun, and K. Obermayer, editors, *NIPS*, pages 1057–1064. MIT Press, 2002. http://books.nips.cc/papers/files/nips15/AA73.pdf.

[9] Young-Hoo Kwon. Dlt method, 2006.

[10] Young-Hoo Kwon. Modified dlt, 2006.

[11] Tim Laue. Eine verhaltenssteuerung für autonome mobile roboter auf der basis von potentialfeldern. Master's thesis, University Bremen, 2004.

[12] A. Matessi and L. Lombardi. Vanishing point detection in the hough transformation space. In *Proceedings of the Fifth International Euro-Par Conference*, pages 987–994, 1999.

[13] Roger Mohr and Bill Triggs. Projective geometry for image analysis, 1996. A Tutorial given at ISPRS.

[14] T. Röfer, B. Altmeyer, R. Brunn, H.-D. Burkhard, I. Dahm, M. Dassler, U. Düffert, D. Göhring, V. Goetzke, M. Hebbel, J. Hoffmann, M. Jüngel, M. Kunz, T. Laue, M. Lötzsch, W. Nisticó, M. Risler, C. Schumann, U. Schwiegelshohn, M. Spranger, M. Stelzer, O. von Stryk, D. Thomas, S. Uhrig, and M. Wachter. Germanteam robocup 2004. Technical report, Humboldt-University of Berlin, University of Bremen, Technical University of Darmstadt, University of Dortmund, 2004. Online: http://www.germanteam.org/GT2004.pdf.

[15] T. Röfer, R. Brunn, S. Czarnetzki, M. Dassler, M. Hebbel, M. Jüngel, T. Kerkhof, W. Nistico, T. Oberlies, C. Rohde, M. Spranger, and C. Zarges. Germanteam 2005. In *RoboCup 2005: Robot Soccer World Cup IX, Lecture Notes in Artificial Intelligence*. Springer, 2006.

[16] D. Schulz, W. Burgard, D. Fox, and A. B. Cremers. Tracking multiple moving targets with a mobile robot using particle filters and statistical data association. In *IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA.*, 2001.

[17] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005. Online: http://www.probabilistic-robotics.org/.

[18] M. Veloso, P. E. Rybski, S. Chernova, C. McMillen, J. Fasola, F. vonHundelshausen, D. Vail, A. Trevor, S. Hauert, and R. R. Espinoza. Cmdash'05. Team report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2005. http://www.cs.cmu.edu/~coral.

[19] T. Röfer W. Nistico. Improving percept reliability in the sony four-legged league. Technical report, 2006.

[20] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR 95-041, University of North Carolina at Chapel Hill, 1995. Online: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf.

List of Figures

1.1 Sensors of *AIBO* ERS-7 12

2.1 BallDeterctionComparison 16
2.2 ct128Schema 17
2.3 ct128Schema 18
2.4 testSetup 20
2.5 systematicError 21
2.6 systematicError 21

3.1 The horizon as calculated by the robot before (a) and after (b) the calibration . . . 24
3.2 Measurement of the calculated vs. the real joint values 24
3.3 DLT: Reference points 25
3.4 Calibration Environment 28

4.1 Dummy particles shot around the ball position as interface to the TeamBallLocator 33
4.2 Rotated Covariance Matrix (Variances represented by purple axes) 34
4.3 x-Position Variance dependent on Distance 35
4.4 y-Position Variance dependent on Distance 35
4.5 Position Variances dependent on Panning Velocity 36
4.6 Velocity Variances 37
4.7 Offset between Percepts and Ceiling Cam distance measurements 38

5.1 The particle field before and after the measurement update. An opponent player is observed. The weight of particles close to the percept is increased. The percept is represented as a gray circle, the particles are black with the size corresponding to the weight. 45
5.2 The particle field before and after resampling. High weight particles are replaced by a corresponding number of low-weight particles. 46
5.3 A particle field before and after a set of measurement update, normalization and resampling steps, exemplifying the information loss in the single filter approach. The measurement update increases the weight of particles close to the percept, the normalisation step reduces all weights to ensure that the weights sum up to 1. Resampling then removes the clusters in the bottom left and right corners. . . . 48

5.4 Measurement step in a multiple filter approach. The percept is only considered in the filter associated with it, the other filters are unaffected. Individual filters are represented by a dashed box. 49
5.5 Creation of a new particle filter. The association of the percept to the single existing filter is very low due to the high distance, so a new filter is instantiated. . . 50

6.1 First step of an ESSE for a 2-dimensional search space; FITNESS(A)=3, FITNESS(B)=1; x and y are the dimensions of the search space. 55
6.2 Different speeds and distances of the behavior “grab ball” 57
6.3 The training procedure for the behavior “grab ball” 58

7.1 WalkPolygons 64
7.2 The Wheel Model for different WalkRequests 65
7.3 Sample measurements of the odometry 65
7.4 Schematics of the odometry table 66
7.5 *RobotControl*: WalkCalibration 68
7.6 *RobotControl*: *WalkShapeViewer* 71
7.7 *RobotControl*: *CustomizeOdometry* 72

9.1 RobotControlXP with some open UserControls 84
9.2 RobotRemote 87
9.3 FieldView 88
9.4 PotentialField with move-to-center and avoid penalty-area 89
9.5 PotentialField with move-to-center and avoid penalty-area 90
9.6 PotentialField-Userinterface 91
9.7 ToolsNumbers 92
9.8 Filter 93
9.9 Horizon Based Tool 94
9.10 Reference Point Based Tool 95
9.11 Gradients 98
9.12 Motion Designer (RC2) 99
9.13 Motion Designer (RCXP) 100
9.14 “EditMof”-View (RC2) 102
9.15 Transition Editor 103
9.16 Button Interface 104
9.17 States of the Button Interface in RCXP 104

10.1 An example placement of the robots for the passing challenge. The circles will be drawn on the field but will not be visible to the robots.[1] 107
10.2 The hard-coded role assignment for the first pass 108
10.3 NewGoal 113
10.4 ZigZagScanline 114
10.5 MainGridScanLines 115
10.6 mergeFragments 117

10.7 freePartOfGoal 118
10.8 The two observers (red) lead the blind blue robot to its destination in the center circle, where a third observing robot is already waiting to confirm the arrival. . . 121
10.9 A simpler approach uses just one observer which tracks the blind robots position. 122

A.1 Linear Function: $range = 50$, $atZero = 20$, x is the distance and $f(x)$ the influence on the field 135
A.2 Parabolic Function: $range = 50$, $atZero = 20$, x is the distance and $f(x)$ the influence on the field 136
A.3 Asymptotic Function: $range = 50$, $atZero = 20$, $solidCenter = 5$, x is the distance and $f(x)$ the influence on the field, for $distance < 5$ the influence is constant $atZero$ 137

List of Tables

7.1 Results of the optimized walktype *WalkWithBall* 70

11.1 The results of the Microsoft Hellhounds at the RoboLudens in Eindhoven 2006 . 124

Index

(1+1)-Evolution Algorithm, 58

Ball Handling in the Passing Challenge, 109
Ball Locator, 31ff
Ball Modeling, 31ff
BallLocator
  Improvements, 38ff
    Distance Offset, 38
    Experimental Forecast Method, 40
    Non-Linearities, 39
    Velocity Filtering, 39
  Kalman Filter, 32
  Particle Filter, 31
Behavior Learning, 53ff
BlindChallenge, *see* OpenChallenge
Button Interface, 102
Button Interface for Manual Award and Mutation Direction, 59

Communication in the Passing Challenge, 110
Comparison of the Optimization Strategies, 61
Covariance Matrices, 33ff

Directionfield, 89
DLT-Method, 25

Energyfield, 89
ESSE, *see* Experimental Search Space Exploration
Evolutionary Strategies, 54
Experimental Search Space Exploration, 55, 59

Fitness Function, 57

Gradient field, *see* Directionfield

Horizon Calibration, 23ff
  DLT-Method, 25
  Vanishing Points, 26

ImageProcessor, 15ff

KickList, 141ff

Learning Methods, 54

mof-files, 75, 100
Motion Designer, 99ff
MSH2006BallLocatorKalman, 32ff

OpenChallenge, 119ff
Optimization of Ball Grabbing Skills, 53ff
Optimization Strategies, 58

Passing Challenge, 105
Playermodel, 43ff
PotentialField, 133–139
PotentialField, 88ff

Reinforcement Learning, 54, 56
RoboLudens, 123
Robot ControlXP, 77ff
Role Assignment for the Passing Challenge, 108

Special Actions, 75, 100

Training Procedure for Ball Grabbing, 58

Userinterface
  FieldView, 88
  Potentialfield, 90