A Behavior-Based Vision System on a Legged Robot

F.A Mantz

M.Sc. Thesis Report
Quantitative Imaging Group
Faculty of Applied Sciences
Delft University of Technology

Supervisors: dr.ir. P.P. Jonker, drs. W.Caarls

February, 2005

Abstract. Knowing where you are is very difficult for robots relying on camera-information. This information is very high-dimensional (many pixels) and highly depends on lighting conditions and the surroundings. These difficulties have resulted in non-robust vision systems, while robustness is essential if we want robots to play soccer in a non-artificial environment.

This report will introduce behaviour-based vision with specific image processor- and self locator- algorithms used at different behaviors; this is a fundamental deviation from the current design architecture now widely used in the RoboCup community, where a general vision system is used: one image processor and self locator working at all times, independent of what the robot is doing.

In this report we will show an implementation of a goalie in the Dutch Aibo Team, that uses behaviour dependent vision. The goalie is able to localise significantly better and under more different lighting conditions than the goalie in the old system (using general image processing and self localization); this while the new goalie was easier to implement, is easier to understand and uses less processing power.

Performance increase is obtained because of the following reasons: first we use a modular image processor; object-specific algorithms with object-specific color lookup tables greatly improve the quality of image processing and robustness against changing lighting conditions. Second we used location information: taking into account where the robot is supposed to be and thus, what he is supposed to see largely increases the performance of image processing. Third: behavior-specific self localization does not have to make the traditional trade-off between robustness and speed. Fourth: A behavior based system only needs a limited set of image-processing algorithms to be active at one time. This reduces the consumption of processing power, which is an important limiting factor in the current system. A limited set of algorithms active per behaviour also greatly reduces the local complexity; both implementation of new concepts and debugging of existing code becomes easier.

Acknowledgements

I would like to thank a number of people without whom this research would not have been possible. At first I would like to thank Pieter Jonker. Pieter has made it possible for me to do my internship at the NEC Personal robot centre in Tokyo. Living in Tokyo has been a most wonderful experience I will never forget. Furthermore, having done research on a robot that didn't compete in the RoboCup competition has given me a head start and great insights about robots useful for my graduation work. Pieter has made all of my research possible providing the robots and all other resources necessary for research. Last I would like to thank Pieter for his guidance, his support and of course for editing my thesis.

I would like to thank Wouter Caarls for helping me out whenever help was needed and for the intensive discussions about my research, which have helped much in clarifying what exactly I was doing.

Much thanks I owe to all members of the Dutch Aibo Team. Andrew, Sylvain and Stefan and the members of the 2nd year software project from Utrecht University I owe much for porting the German Code in such a short time. Thanks to their enthusiasm and effort, I was in possession of a "working" soccer playing robot-team only 2 months after I started my research and I could start doing real research in a very early stage. Second I would like to thank the Dutch Aibo Team for giving meaning to my graduation work. Being part of an actually competing RoboCup team has kept me motivated for my whole graduation year. The RoboCup competitions in Paderborn and in Lissabon have been wonderful experiences I would never have wanted to miss.

My fellow Masters students, especially the ones from the robot room, I would like to thank for their company and the interesting discussions about other fields in robot research. All my friends in Delft I would like to thank for sharing my time as a student.
My mother, father and sister I would like to thank for all the support you have given me in the past 24 years.

Definitions of terms and variables

High level system
High level means having a high (symbolic) abstraction level in the decision process. Words as ball, opponent, field, near, etc. are abstractions. For example, a high level thinking loop thinks in terms of: *If opponent near ball, then move ball away from opponent*.

Low level system
Low level means having a low (symbolic) abstraction level; close to the real world and real sensor and actuator values. An example of a low level decision loop is: *if distance sensor s is smaller than x, then move wheel y*.

Process
A process is an independently running piece of software that can basically run (quasi) in parallel with other processes. If two different processes want to share information, they need some sort of communication path, e.g. shared memory, message passing systems; different processes compile to different object files (*.obj).

Module
A module is a piece of source code that is internally denser interwoven then it has connections with the rest of the software. Its connections with the other software (also in modules) is explicitly specified (the module interfaces). The source code of big software projects is divided in several modules because it makes the software understandable for software designers and programmers. The use of different modules does not only make a project understandable, it also allows for cooperation between several designers, because different designers can work on different modules independent from each other. Different modules do not necessarily compile to different files. Modules that are closely related (e.g. the vision modules) can be grouped together (and compiled) in one source file.

Vision
The interpretation of information from the image forming sensors. In our system the vision system consists of the perception (image processor, goal detector, ...) and the object modeling subsystems (self localization, obstacles modeling, ball modeling,…).

Image processor
This subsystem searches images acquired from the camera, for objects that are known to exist in the system's world. The image processor outputs percepts.

Percepts
Recognized known objects in the system's world, such as lines, goals, flags, opponents.

Self localization
The generation of the robot's pose using the found percepts.

Robot pose
The position of the robot in the playing field. The robot pose consists of his absolute position, the translation (x,y) and of his orientation (θ).

True / false positive
A detected object in the image is correctly labeled as one of the objects the system knows (true positive), or incorrectly labeled as an object the systems knows (false positive)

Correct / false acceptation rate (CAR / FAR).
The number of objects detected and labeled (accepted) in an image where they actually were (Correct) or were not (False) present.

Update rate (UR)
The update rate indicates the impact of single percepts in the self localization process.

Quality of self localization (Q)
The symbol Q indicates the quality of the self localization. If Q is high, the robot is likely to know where he is.

Number of objects searched for (N)
N is the total number of objects searched for by the image processor.

Number of objects visible ($N_{visible}$)
The number of objects actually visible in a camera image

Table of contents:

1. Introduction

1.1. Robot research

Research in autonomous robots
The research in autonomous robots can roughly be divided into two main streams: behavioral and functional robotics.

- Behavioral robots are often simple autonomous robots that may display complex behaviors. Examples are the Breitenberg Vehicles. For all information about this interesting field of robots we refer to Pfeifer and Scheier [9].
- Functional robots are robots designed to do a specific (complex) task. These robots are designed not to show intelligence, but to do useful things, bounded by its specific task. They often use complex algorithms.

This report can be seen in the field of functional robotics. We can roughly give a design of how the architecture of a functional robot looks like. The robot receives information from the world (it senses), decides what to do (it thinks), and does something (it acts).

For each of these three parts, many different implementations are possible. The final solution for one part is mainly determined by the purpose the robot has to serve.
Different research fields traditionally only cover one of the parts of the robots. In figure 1.1 a representation of an autonomous robot as a sense-think-act loop is given, with examples of solutions and related fields of research.

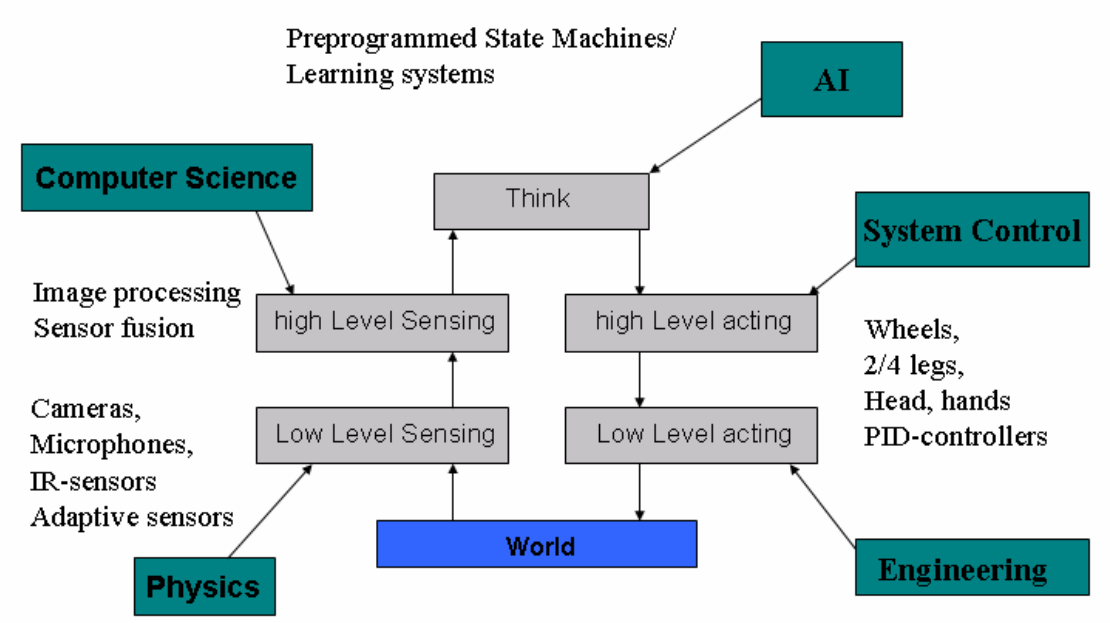


Figure 1.1. An autonomous functional robot represented as a sense-think-act loop. Examples of solutions for the separate parts are written next to the parts. The green blocks show the research fields traditionally covering the separate parts.

The RoboCup competition
In order to promote AI- and robotics- research and to stimulate exchange of knowledge in the different research fields, the RoboCup [1] Initiative was founded.
In the RoboCup competition, teams of robots compete in games of soccer in several leagues; there is the humanoid league (walking robots), the middle-size (autonomous wheel driven

robots), the 4-legged (Sony Aibo's), the small-size (small robots, driven by a central computer) and the soccer-simulation league.

The bottleneck in performance of the robots differs per league. For humanoids, controlling the motion (walking) currently is the most difficult part. Sensing is the most important part in both the middle-size and 4-legged league. Artificial Intelligence is most important in the small-size and simulation league.

Besides competing in playing soccer, there is also a robot-rescue league. The robot has to rescue persons in either a simulated or a (limited) real world scene.

The Dutch Aibo Team

The research described in this report is done as part of the Dutch Aibo Team [2], cooperation between (in alphabetic order) the universities of Amsterdam, Delft, Twente and Utrecht and the Decis-Lab. The Dutch Aibo team competes in the 4-legged league. It was founded in the beginning of 2004 and has performed with reasonable success at both the German Open 2004 [3] and the Lissabon 2004 world championship. The Dutch Aibo team pre-qualified in Lissabon for the RoboCup world championship in Osaka 2005.

1.2. Behavior-dependent vision

Importance of the vision system

At first glance one might underestimate the importance of vision research. The reason for this is that one doesn't win a soccer game with a vision system; one wins it with motion and behavior control. The robots that walk the fastest, kick the hardest (motion) and play the smartest (behavior) will win. Being good in seeing things doesn't make you score goals. Having a working vision system is only a requirement for being able to play any soccer, as having a working behavior control and working walking engine are requirements for playing soccer.

The reason why vision-research is such a crucial factor in the robot research is its complexity. Image processing is far more complex than motion and behavior control. Motion works with approx 20 parameters (the joints), Behavior control works with less then 50 parameters (e.g. the position and may be orientation of the objects).

Vision for an AIBO works with approx. 30 000 inputs: the 208*160 camera-pixels. The human vision system even has an equivalent of 10 million pixels. Even though these inputs are highly correlated (big objects cover many pixels), the number of effective parameters is much bigger than those of motion and behavior.

Understanding operations on all these parameters is conceptually difficult. Even more important, operations on all these parameters require significant processing power, limiting the amount of possible operations.

The requirements of the vision system are also much higher than those of the behavior- and motion control systems. Behavior is designed in a virtual world, with only positions as input, making designing very easy. The design of motion already is slightly more difficult; it has to take into account the characteristics of both the actuators and the playing field, which can vary for different robots and different fields.

Vision however has to cope with the visible environment, which is far from constant in the real world. The way a camera image captures an object from the real world depends e.g. on camera characteristics, such as the amount of light and the lighting-temperature. In the real world, lighting conditions can be very different for different situations: both in one game (sunspots or shading) and between different games (outside vs. inside light). A desired vision system is able to cope with all these different situations.

Pre-requisites of the vision system

During tournament games, lighting conditions are kept very artificial: a white wall surrounding the field prevents the robots from seeing objects other than belonging to the field. Many flood-lights, placed above field, provide a large quantity of illumination, largely uniform divided over the playing field, and not changing during the tournament. Under these circumstances, the current AIBO vision system is good enough for playing soccer, and the winning team is the team that has the best behaviors, the fastest walks and the best kicks.

If circumstances are only a little less than tournament-conditions, the current system fails on localization and the robots are not able to play soccer. The sense-part then becomes the bottleneck in the performance of the robots.

During a practice match of the Dutch Aibo Team at Nemo, where no white wall was surrounding the playing field and the amount of available light was not abundant, it even proved impossible to make a color table sufficient for playing any soccer. The robots detected far too many objects where they didn't exist, or failed to detect enough objects where they did exist. The goalie was not able to localize well enough to stay in its goal; the players could not localize well enough to kick the ball in the correct direction.

If we want a robot team that can localize during practice matches and that will not be obsolete in two years when the tournament rules will be less artificial, significant improvement of the robots vision system is required.

Robustness of the vision system

The first logical attempt in improving the vision system, is improving the algorithms for its principal components: the image processor and the self locator. Much research has been and is being done on building shape-based image processors [4] on top of color-based image processors, building auto-adaptive vision systems robust against changes in lighting conditions [5], and building self locators that fuse the information from multiple robots [6].

These fields of research are very valuable and do result in important improvements in parts of the vision system. These results, however, only solve small parts of the problem with robustness, and do not result in the dramatic performance-increase we require. These new improvements require more processing power, while processing power is limited. For really solving the robustness problem, a fundamental different approach for vision is necessary.

In the middle-size league, such a fundamental improvement has come in the form of omni-vision. Robots use a camera that allows the whole of the playing field to be seen at all time, greatly reducing the complexity of the vision problem. This solution though is not available for the Aibo league, which doesn't allow hardware changes. There is also another reason why we want to look beyond omni-vision: humans don't have an omni-directional vision system, (as they don't have laser detectors, nor a GPS-system, nor are all connected to one central computer); it is very unlikely that robots will be allowed to use an omni-directional vision system in 2050; it is merely a temporal solution.

A fundamental improvement in the vision-system is an absolute condition for both the ability to play Aibo-soccer in a few years and to play overall Robosoccer in 2050.

Behavior based vision

This report a behavior based vision system, a new approach that hopefully enables the required improvement of the image processing and self localization. In this approach, we stop looking at the robot system as one big sense-think-act loop (fig 1.1.), with one algorithm for image processing and one for self localization. Instead, we break the system down in several sense-think-act loops, with different algorithms for image processing and self localization; in as far as they belong to different behaviors.

This new approach will allow for a better vision for four reasons. (See figure 1.2).

1. Modular Image Processor.

We can improve the performance of the image processor if we use separate methods for detecting separate objects. Using specific color lookup tables and scan-lines for the detection

of different objects (blue-goal, yellow-goal, lines, etc.) instead of using one general color-table for the detection of all objects, significantly increases the performance.

2. Location Information Image Processing.
Behavior dependent vision implies location-dependent vision, which can improve the performance of the image processor. Only searching for objects in places where they are expected to be seen, can highly reduce the number of false detections.

3. Behavior Specific Self Localization.
The odometry error, the reliability of the robot-pose and the performance of the image processor, all depend on what the robot is doing. With behavior-based vision we can take these factors into account when designing different behavior specific self locators. Trade offs between speed and robustness are less necessary and a higher performance is possible.

4. Hierarchical Behavior-Based Software Architecture.
With a behavior-based software design, because of the location dependence, a specific behavior will only require a limited set of vision algorithms to be active at one time. Consequently, more processing power will be available for these individual algorithms. For one specific behavior, only a limited set of image processing- and self localization algorithms will be active, thus the complexity of the different sense-think-act loops will be significantly lower. A designer needs to understand less code in order to be able to understand what exactly happens in a certain behavior. In combination with a clear and logical hierarchical behavior-tree, this will result in a robot system in which it is easy to understand what really happens. Understandable software is crucial for development.

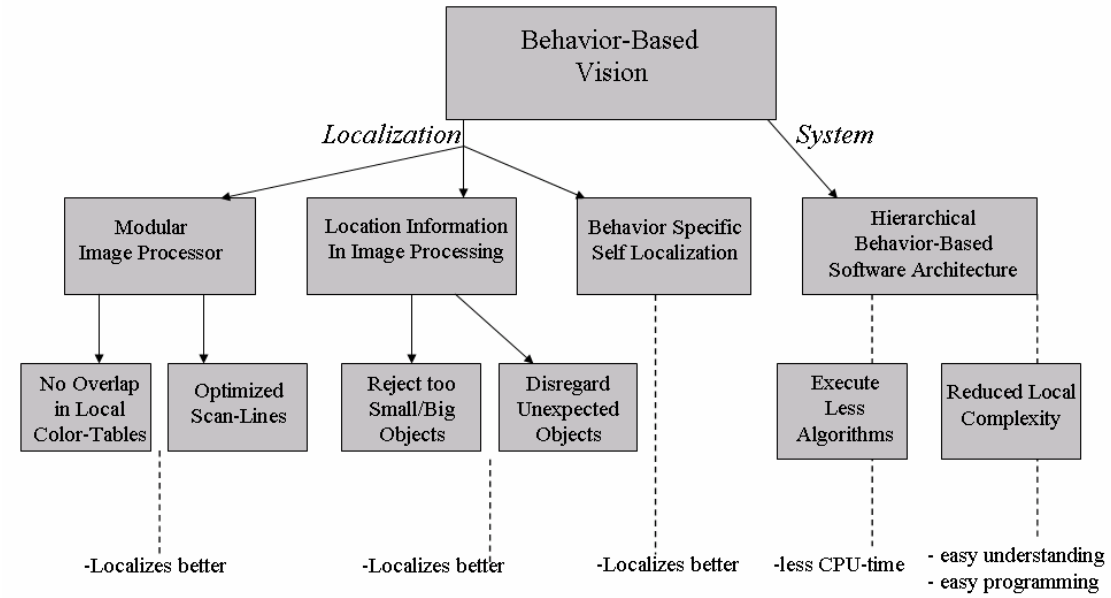


Figure 1.2. The advantages of using behavior-based vision.

In this report we will give the theoretical arguments why behavior-based vision leads to a better (more robust) vision system. We also will describe the results of an implementation of this system. We have built a demo of a goalie for the Dutch Aibo Team, using a behavior-dependent vision system. We will describe its architecture and the algorithms used for the goalie, and will show through a number of tests that this goalie can localize more robust than the goalie of the old system (using a single general vision system). This in addition to the fact that the new goalie was easy to implement, is easy to understand and requires less computation time.

1.3. Guide through this thesis

In chapters 2-4, the architecture of the Dutch Aibo Team 2004 will be briefly described. Chapter 2: *Hardware: ERS-7 specifications* covers the hardware specifications of the Aibo Robot. Chapter 3: s*oftware architecture* covers the overall software architecture of the Dutch Aibo Team. Chapter 4: *main algorithms* gives a small description of the approaches used in the most essential parts of the soccer-playing robots: the *image processing* (4.1), the *self localization* (4.2), the *behavior control* (4.3) and the *motion* (4.4).

In Chapter 5: *Estimating the quality of self localization* we will argue which parameters drive the performance of a self locator. We will show that the quality of self localization not only depends on the quality of the image processing algorithms, but also is highly location- and behavior dependent.

In Chapter 6: *A behavior based vision system* we will describe how the different implications of behavior-based vision influence performance: using *modular image processing* (6.1), using *location information in image processing* (6.2), using *behavior-specific self localization* (6.3) and using a *hierarchical behavior based software architecture* (6.4).

In Chapter 7: *Behavior based vision software architecture* we will describe how we have implemented behavior-based vision in the DT2004 architecture. We will describe how solutions for image processing and self localization are controlled from behavior control.

In Chapter 8: *Behavior-specific algorithms*, some algorithms designed specifically for the goalie with behavior-based vision will be described.

Chapter 9: *Results* shows the results of some tests comparing the performance of the old- and the new-goalie.

Chapter 10: *Conclusions* presents the most important conclusions.

Chapter 11: *Discussion and recommendations* discusses which parts of the behavior based vision system, designed for the goalie, is applicable for the other players (11.1). Also it will be suggested how one can further continue with developing the behavior based vision system (11.2). Also it gives a set op possible interesting topics for further AIBO research (11.3).

2. Hardware: ERS-7 specifications

This section covers the hardware specifications of the AIBO ERS-7 that is used by the Dutch Aibo Team in their robot soccer games.

CPU:
64-bit RISC processor, 600 MHZ

Memory
64 MB Ram

Image sensor:
350.000-pixel CMOS image sensor. Functional 208*160 * 3 channels (y,u,v)

Integrated sensors
Infrared sensors x 2 (nose + chest)
Acceleration sensor
Vibration sensor

Audio input
Stereo microphones in the ears

Other input sensors:
1 Head sensor
3 Back sensors
1 Chin sensor
4 Paw sensors

Degrees of freedom:
Head: 3 (tilt, pan, tilt2)
Mouth: 1
Legs: 4 x 3
Ears: 1 x2
Tail: 2

Audio output
Speaker in the mouth

Communication
Wireless LAN IEEE 802.11b (2.4 GHz)

Dimensions and weight:
180x278x319 mm (w x h x d)
1.6 kg.

Power
Energy station for charging battery: 2 hours charging time.

Program media
16 MB Aibo memory stick.

3. Software architecture: process view

The software architecture described in this chapter was used by the Dutch Aibo Team during the RoboCup 2004 tournament in Lisbon. It was developed by the German Team and partly adapted by the members of the Dutch Aibo Team. For a more detailed description we refer to the German Team Report GT2003 [7].

In this chapter we will describe the architecture from a process viewpoint. What tasks are performed by the robot, how are these tasks divided in different modules and processes, and how information is transferred between these modules and processes.
This chapter is concerned about what happens in real-time: through what steps does a new input (e.g. an image) lead to a new output (e.g. an actuator command).

3.1. Modules, solutions and processes

A robot playing soccer performs many tasks. It looks at the world, determines where it is, locates opponents and obstacles, determines what action to do next, it walks and kicks. In this architecture, playing soccer is divided in about 20 tasks. The algorithms for performing a certain task are grouped in a module.

3.1.1. Modules

A module is a piece of software, with a well defined interface. It contains an algorithm that determines in what way an input, or set of inputs, given to the module, leads to an output.
In figure 3.1 one can see an example of such a module: the image processing module.
The image processor fulfills the task of generating percepts using acquired images and the camera matrix. Its inputs and outputs make up the interface.

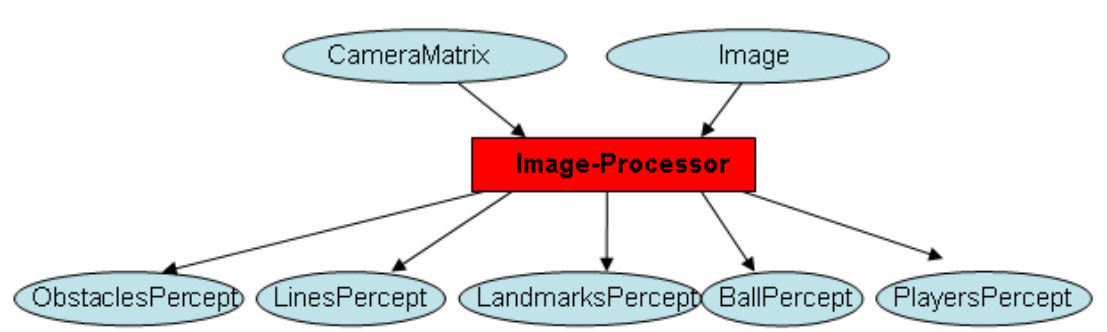


Figure 3.1. The module image-processor, with the camera matrix and an image as inputs. The algorithms in the image-processor determine the percepts at the output.

3.1.2. Solutions

A solution defines the algorithms in a module. It defines the way in which an input, or set of inputs, lead to a set of outputs. If there is more than one solution available for one module, it means there are different algorithms available, each resulting in a different performance.

For example, for the image processor, different solutions can be used. One can use a color table and find percepts by only matching color values. Or one can use a method that uses shape to detect these same percepts.

3.1.3 Switching Solutions

The German Team has designed the software in such a way that it very easy to switch between different solutions for one module. The source code and the compilation of the

DT2004 project can contain several solutions for one module, with different algorithms, but with the same interface. Solutions can be switched in two ways:

1. In a file (modules.cfg), one can tell what solution has to be used for a module.
2. Solutions can be changed at runtime through robot-control.

Robot-control is unavailable during actual games, and one can only set the file modules.cfg once, prior to the match. This means that during the game there is only one solution available for a module. The algorithms in the module don't change during the game. Since there is only one solution active, and we can know the algorithms in this solution, we can always know how a certain set of inputs leads to a set of outputs.

3.1.4 Solutions for behavior control

The working of the module behavior control is different from that of all other modules. For behavior control, there is not just one solution running during a soccer match, determining in which way outputs are generated from the inputs. Instead, behavior control incorporates many different behavior modules (fig. 3.2). Each of these behavior modules can be seen as a different solution for the module behavior control. The outputs generated by behavior control cannot straightforwardly be predicted from the inputs. They depend on the *state* of the behavior control, i.e. the behaviors that are currently active. The active behaviors change all the time during a soccer match.
Chapter 4 explains in slightly more detail how behavior control works.

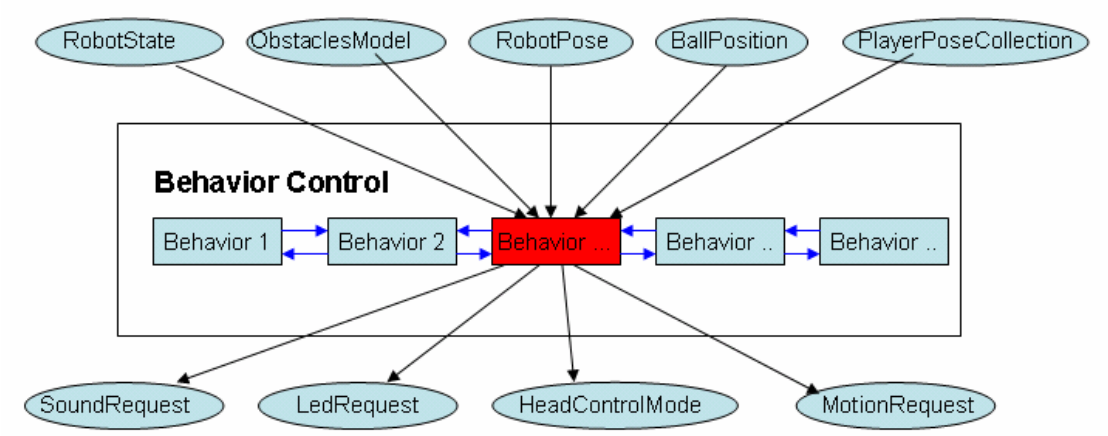


Figure 3.2. The module behavior control. Behavior control consists of many behaviors, the current behavior being used changes during the game. The way the outputs (motion requests) are generated from the inputs (mainly robot-pose), is different for the different behaviors.

3.1.5 Overview of all tasks

In Figure 3.3 you can see all the tasks that can be used for playing soccer. In this figure the rectangles indicate Modules, arrows and circular objects represent the interfaces. They indicate the information used by (input) and produced by (output) a module.
The four modules most crucial for the performance of the robot and containing the least straightforward algorithms are displayed in red.

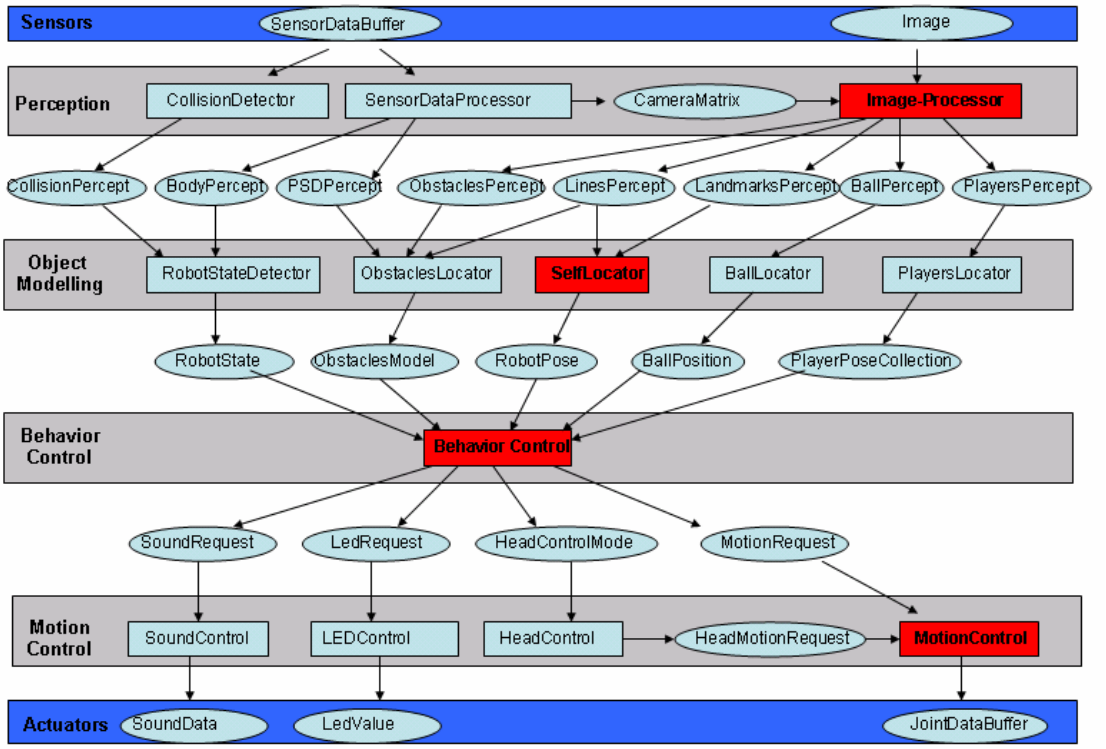


Figure 3.3. The different tasks in the Dutch Aibo Team 2004 being performed by the robot.

A more simplified representation of figure 3.3 can be found in figure 3.4. In this the robot system is represented as one sense-think-act loop. The arrows indicate the information that flows. Perception, object modeling and behavior control are integrated in one process (cognition). Motion control is executed as independent process (motion), as are the sensors and actuators (robot).

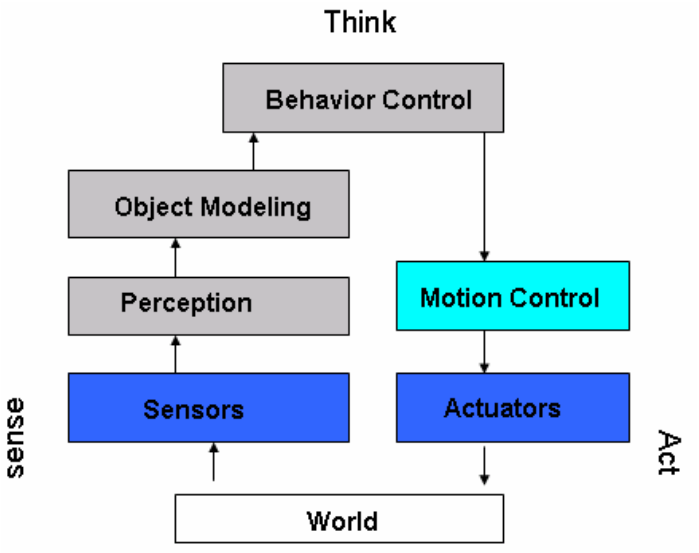


Figure 3.4: the design architecture represented as one Sense-Think-Act loop.

3.2. Process layout Dutch Aibo Team

In Figure 3.3 one can see the different tasks performed by the different modules, and the information that flows from one module to the next. Although one can represent this whole

process as one sense-think-act process, the robot does not actually work as one loop in one process. The tasks are divided over several processes, running in parallel. The perception, object modeling and behavior are all integrated in one process. Motion control is in another.

3.2.1. Why use different processes

The main reason for using more than one process is that it allows for reserving a constant amount of processing power for tasks that need to be performed many times a second and at a constant rate. Calculation of motor commands for walking, requires relatively few CPU cycles, but needs to be performed at full frame rate to make sure the robot walks smoothly. The processing of images, however, requires so many CPU cycles that the image processing can only be done in about 30 times a second. Since we don't want motion control to have to wait for the image processing, they are located in separate processes.

3.2.2. The processes in DT2004

In our architecture, 4 independent process loops are used, as can be seen in figure 3.5. All the algorithms for different tasks as previously identified are divided over 2 processes. One for processing images, self localization and behavior control (*Cognition*). One for calculating motor commands (*Motion*). The third process is running on the robot, grabbing images and controlling motors (*Robot*). The fourth process is one for debugging purposes (*Debug*). It collects and distributes messages sent through message queues from and to other processes and the PC. The process Debug is only active during development; it is inactive during actual RoboCup games, and therefore not displayed in the design architecture.

A schematic representation of the different running processes and their interaction can be found in figure 3.5. The arrows indicate the packages that are sent between the different processes.
Cognition receives images and other sensor data from robot; Robot receives motor commands from *Motion*; Motion receives motion requests from Cognition and sends the motions currently being executed back to Cognition; Debug messages such as images, color tables, percepts and joint values can be sent back and forth to a computer through the process Debug.

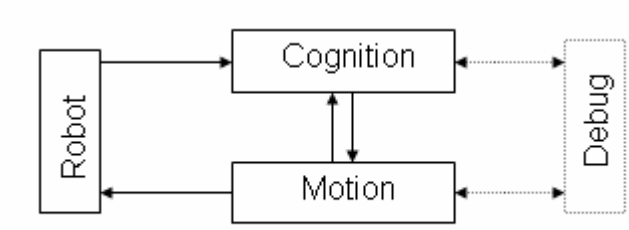


Figure 3.5. The process layout used by the Dutch Aibo Team.

Note that the process layout of the Dutch Aibo team is much simpler than the process layout of e.g. Clockwork Orange [14], a team of Delft University competing in the Middle Size League. In the Dutch Aibo team, all processes that depend on new image input (Image processing, self localization, behavior control etc.) are all integrated in one process (Cognition). In the Clockwork Orange (CO) architecture, the layered module hierarchy consists of the modules{Communication, Motion, Kicker, Vision, Sound} on the virtual devices layer, {Behaviors and World Knowledge} on the Robot Skills Layer, and {Team Skills and Mission Manager} on the Top Management Layer. Each module is at least a separate process; some large modules consist of more parallel running processes, like the vision module and behaviors module.

When one compares the CO with the DA architecture, the CO modules {Communication, Motion, Kicker, Sound} form the DA module Robot. The CO modules {Vision, World Knowledge} form the DA module Cognition and the Behaviors Module is equivalent to the DA Module Motion. As the DA team has no specific team behavior, there is no Team Skills software present in the DA team at this moment. In the CO team the robots actively coordinate the fulfilling of their attacker / defender roles. The Mission Manager module of CO is comparable with the Debug module of DA, albeit that the mission manager also takes care of the system integrity of the robot and reanimates the robot if the software somewhere goes astray (e.g. hangs).

3.2.3. Communication between modules and processes

Within one process
The communication between different modules within one process is straightforward. All modules are executed sequentially, and work in memory is available to all modules.
The data that can be used by and updated by a module is defined by the module's interface.

For example:

1. The image processor processes an image and calculates percepts. These percepts are stored in a common memory as defined in *Cognition.h*.
2. When this is done, the self locator is executed, using these same percepts as input and updates the robot-pose.
3. When this is done, behavior control uses the robot-pose as input to calculate the new motion request.

Between processes
The inter-object communication, that is the communication between tasks in different processes, is performed by *senders* and *receivers* exchanging *packages*. A sender contains a single instance of a package. After it was instructed to send the package, it will automatically transfer it to all receivers as soon as they have requested the package. Each receiver also contains an instance of a package. The communication scheme is performed by continuously repeating three phases for each process:

1. All receivers of a process receive all packages that are currently available.

2. The process performs its normal calculation, e.g. image processing, self localization, etc. During this, packages can already be sent.

3. All senders that were directed to transmit their package and have not done it yet will send it to the corresponding receivers if they are ready to accept it.

Note that the communication does not involve any queuing. It resembles a shared memory approach. Whenever a process enters phase 2, it is equipped with the most current data available. For a more detailed description on senders and receivers, please read chapter 2.2.3 and Appendix C of The German Team description [7].
Note that two different processes are independent loops with different update rates. When *Motion* finished its loop and has sent motor commands to the robot, but has not yet received a new package from *Cognition*, it will not wait for a new motion request (e.g. walk, stand, special-action), but will calculate new motor values based on the old motion request.

4. Software architecture: algorithms Dutch Aibo Team

This chapter covers the essential algorithms of the DT2004 software: the image processing (4.1), the self localization (4.2), the behavior control (4.3) and motion control (4.4). A summery of the algorithms used in the complete system is drafted in figure 4.1.

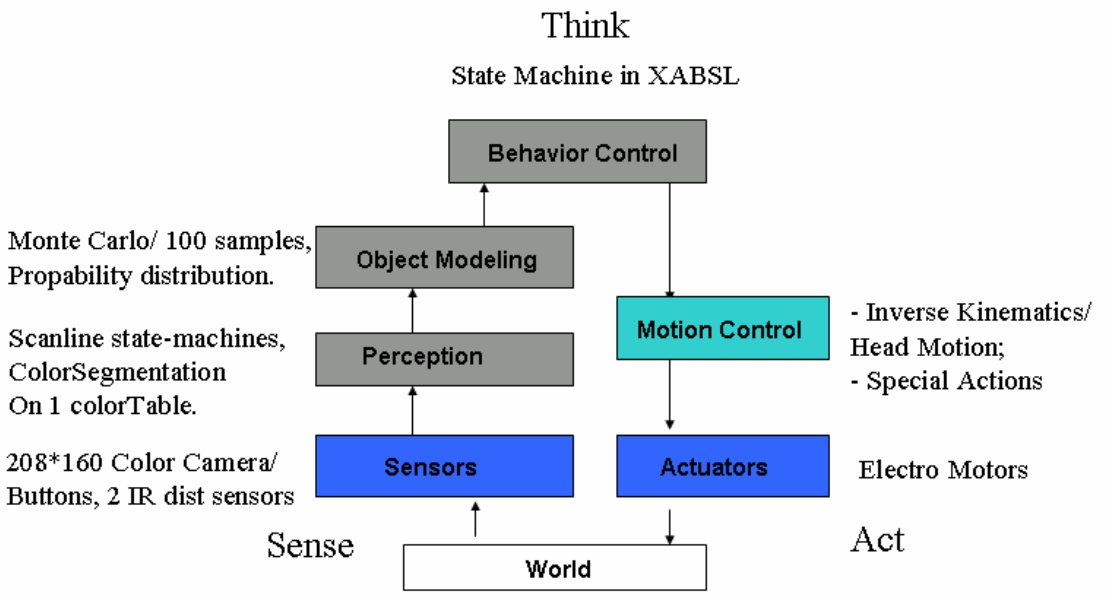


Figure 4.1. The sense-think-act loop of the DT2004 software, with the solutions used for the individual parts.

4.1. Image processing

The image processor takes the camera image and the camera matrix as input; it searches for percepts, (flags, goals, lines and ball) in the images. The image processing is mainly based on color-segmentation followed by shape evaluation. Not all pixels in the image are used for image processing, only the pixels positioned on a limited number of scan-lines. The core of the image processing is only 1-dimensional!!

4.1.1. Using a color lookup table

A camera image consists of 208*160 pixels. Each of these pixels has a three-dimensional value p(Y,U,V). Y represents the intensity; U and V contain color-information; each having an integer value between 0 and 254.

In order to simplify the image processing problem, all these 254*254*254 possible pixel-values are mapped onto only 10 possible colors: white, black, yellow, blue, sky-blue, red, orange, green, grey and pink, the possible colors of objects in the playing field.

This mapping makes use of a color-table. The color-table is a big 3-dimensional matrix which stores which pixel-value corresponds to which color. A good tool for calibrating this color-table is available in the *Robot Control*. (Appendix B, German Team Description [7])

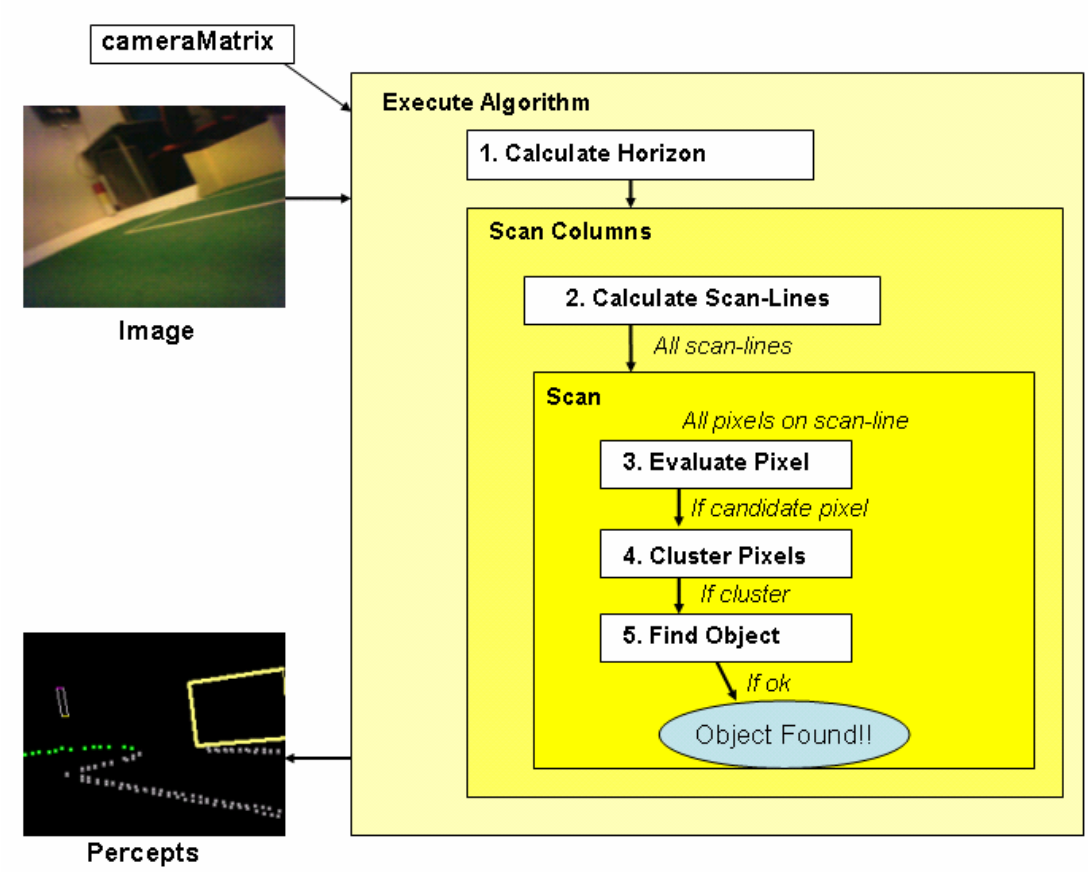


Figure 4.2. A schematic representation of the image processing.

4.1.2. Main image processing algorithm

The main algorithm for image processing can be found in figure 4.2. The image processor works on a camera image. This camera image is send together with the camera matrix. The camera matrix represents the angles of the head-joints at the time the image was captured. Percepts in an image are found through the following steps:

1. Calculate horizon

At first, the coordinates of the horizon are calculated. The horizon represents the projection of what the robot would see at eye level if it would look straight ahead (Fig 4.3a). Details on the calculation of the scan line can be found in chapter 3.2.1 of The German Team Report [7].

2. Calculate scan lines

Images are scanned on scan lines perpendicular to the horizon. In the function scanColumns(), the desired amount of scan lines is determined with the according start and end coordinates. An example of calculated scan lines for flag detection can be seen in figure 4.3b.

3. Evaluate pixel

Every scan line, with start and end coordinates, is scanned for points possibly belonging to an object. This scanning process is a state-machine; whether a pixel is a possible candidate depends both on its color and on the colors of the pixels previously scanned. A certain amount of yellow pixels, followed by a certain amount of pink pixels, followed by a white pixel, indicates a possible yellow/pink flag. The color-table is used to determine the color of each pixel (fig 4.3e).

4. Cluster pixels

If the point is near an existing cluster, it will be added to the cluster; else a new cluster (possible object) will be created.

5. Find object
Each cluster indicates a possible object. All the points in such a cluster are used to determine whether or not the candidate truly is an object or not. This decision process uses object-information such as the height and width of objects. For each detected object, information such as the size and position relative to the robot are determined. For some objects, such as flags, additional algorithms are used for determining the characteristics of possible objects. An example of this can be seen in figure 4.3c. If the object is accepted, it is added to the percept collection (fig 4.3d).

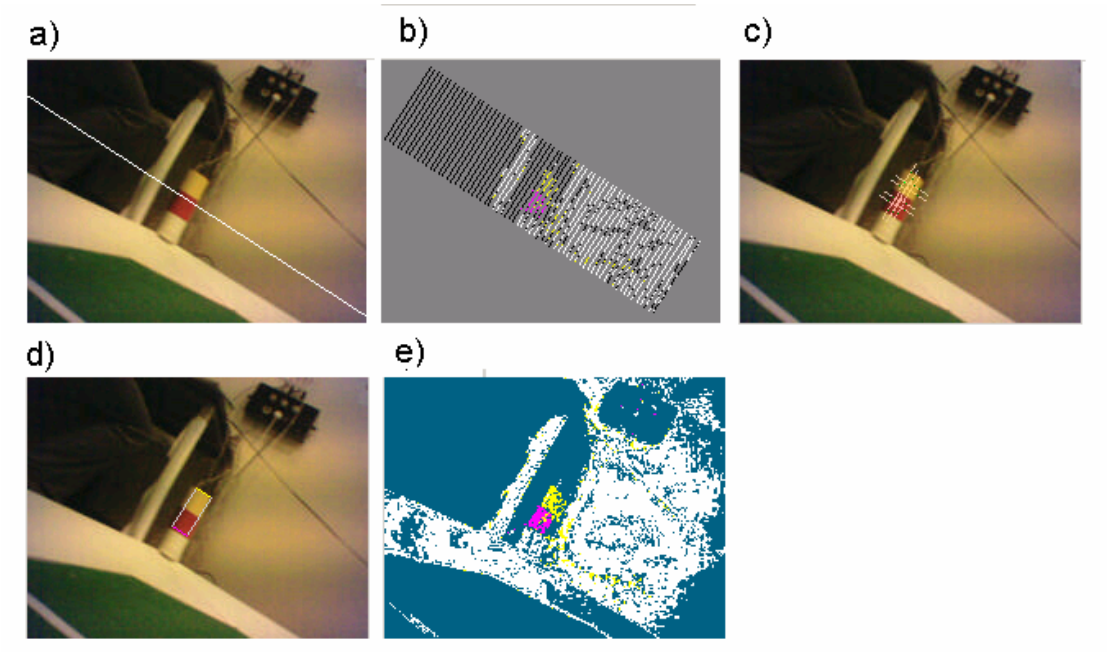


Figure 4.3. Image processing on images. a) The horizon. b) The scan-lines calculated perpendicular to the horizon. C) Additional scan-lines for both classification and the determination of width, height and position of a possible flag. d) Flag detected. E) Image as seen by the color-table.

4.2. Self localization

The self localization of the robot is the ability to determine their location from sensory input. The approach used in the Dutch Aibo Team is *particle filtering* or Monte Carlo Localization [11] , [12]. We have used the implementation of German Team [7], [13]. In this chapter we will briefly summarize the self locator in the Dutch Aibo Team.

4.2.1. Robot-pose from 100 samples

The self localization is the process of obtaining the robot's pose (x,y,θ) from the found percepts. The Self localization in the Dutch Aibo Team is a probability-based method that keeps track of 100 candidate robot-poses (black arrows in figure 4.4), called samples. The actual robot-pose (yellow robot in figure 4.4.) is obtained by averaging over the largest cluster of samples.

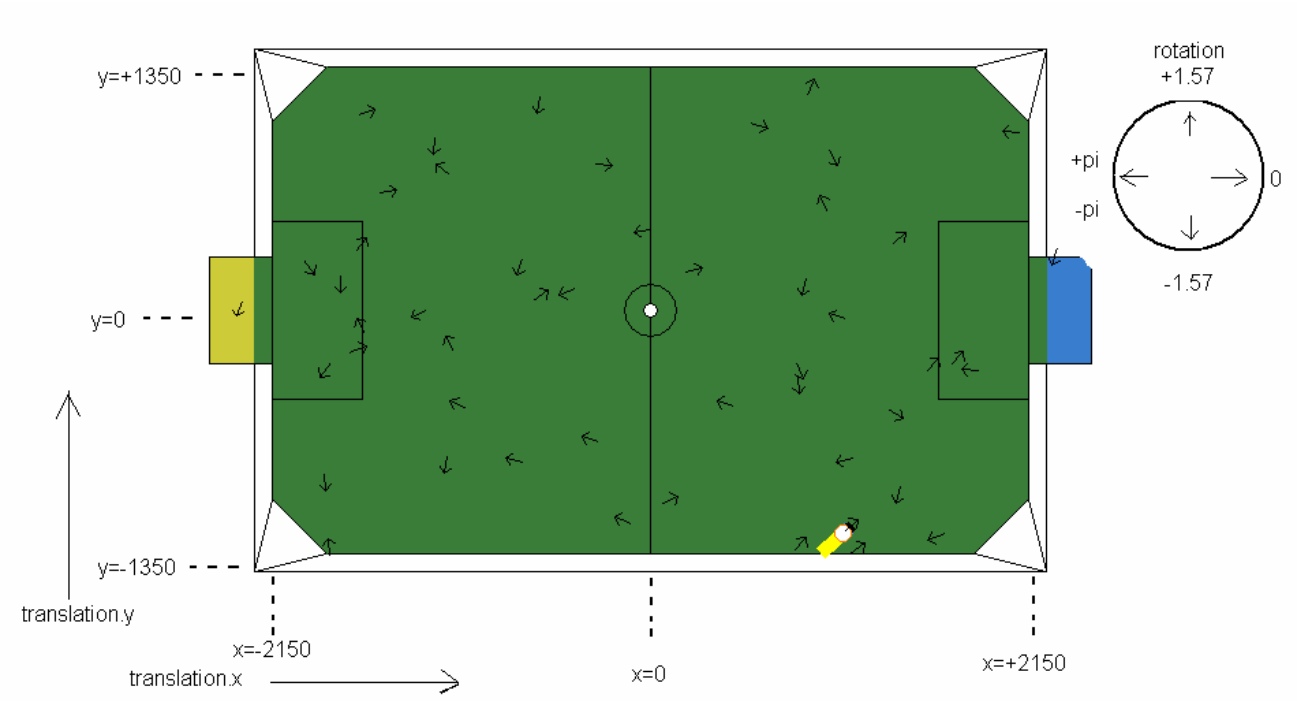


Fig 4.4. The self localization at initialization; 100 samples are randomly divided over the field. Each sample has a position x, y, and heading θ in absolute playing-field coordinates. The robot's pose is evaluated by averaging over the largest cluster of samples.

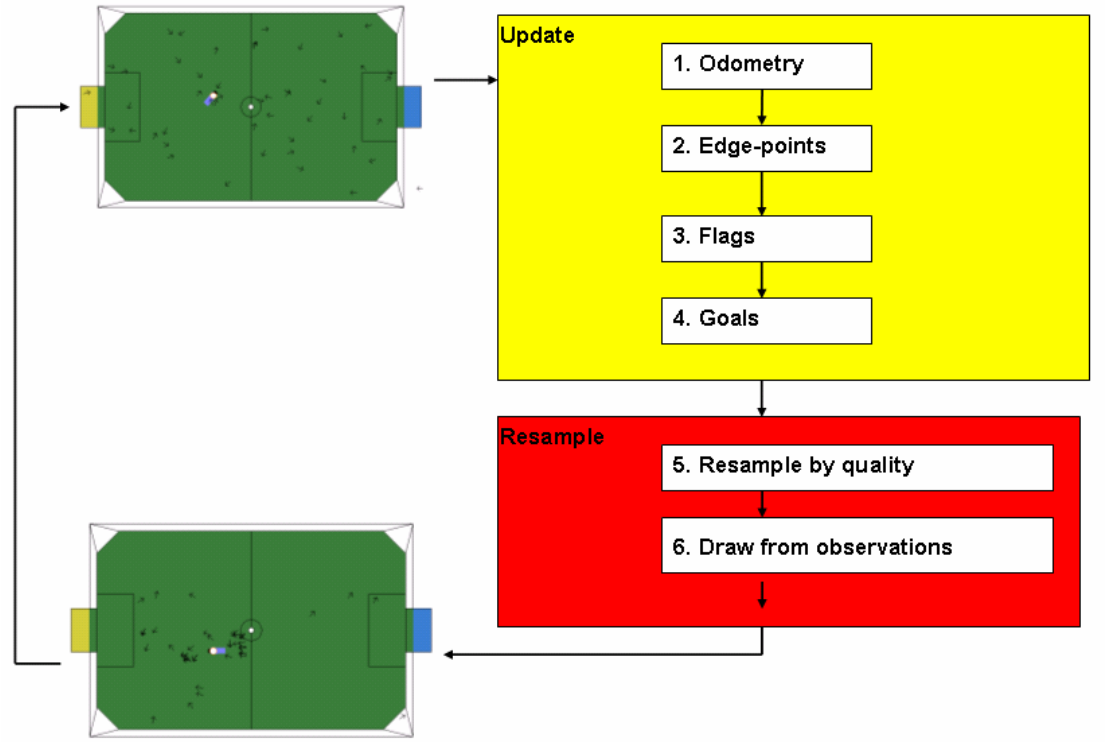


Figure 4.5. Self localization loop. Every time-step, first the 100 old samples are updated, i.e. their coordinates can be slightly adjusted and their qualities changed with a limited amount, by comparing the samples with the received percepts (lines, flags, goals).
In the next step (Resample), high quality samples are likely to be attenuated and low-quality samples are likely to be removed. Very low-quality samples can be replaced by samples directly drawn from observations (e.g. from the intersection of 3 flags).

4.2.2. Calculating the samples

The algorithm for updating the samples is shown in figure 4.5. The algorithm has some places where it updates the coordinates of the samples, but it mainly updates their qualities. For each sample, 5 independent qualities are monitored; one quality for each edge-point type (field line, border, blue- and yellow goal), and one quality for the other percepts (goal, flag). The total quality is obtained by multiplying the 5 qualities.

1. Odometry
The first step in updating the existing samples is by adjusting to the odometry offset $\Delta_{odometry}$. If the robot is known to have walked a certain distance and heading, all the samples are displaced with the same distance and heading. In this step a little random error is added to each sample, to model the error in the odometry.

2. Edge-points
After the samples are updated by the odometry, the samples are updated by the measurements of distances to the edge-points. One edge-point is randomly chosen for each edge-type (field line, border, blue- and yellow-goal). For every sample, the modeled distance to an edge-point is found in a lookup table. The sample is updated by comparing the modeled distance with the measured distance in 2 ways:

- Adjusting the translation. The difference between the measured and modeled distance between a sample and an edge-point gives a metric deviation of the robot's posture. This is used to slightly move the translation of the sample in a way that this difference is reduced
- Adjusting the quality. If the difference between the measured and modeled distance is large, the quality of a sample is decreased. If this difference is small, the quality is increased.

3. Flags
In this step the perceived flags are used to update the qualities of the samples. If a flag is seen, the quality of a sample is updated by evaluating the difference between the angle at which the flag is seen, and the angle at which the flag is expected to be seen.
The quality q of a sample is calculated as the similarity s of the measured angle and the expected angle by applying a sigmoid function to the difference of both angles.

4. Goals
In this step the perceived goals are used to update the qualities of the samples. The front posts of the goals are used as the points of reference. The qualities of the samples when a goal is seen are obtained in exactly the same way as when flags are seen.

5. Resample by quality
The quality of each sample is evaluated by multiplying the 5 individual qualities. These evaluated qualities are used for copying the samples from the updated distribution to a new distribution. More probable samples are copied more than less probable ones, improbable samples are removed.

6. New samples drawn from observations
In a second step, some very low-quality samples are replaced by so-called candidate postures, these are samples with coordinates analytically evaluated from a combination of percepts. Two methods are implemented for calculating possible robot postures. They are used to fill a buffer of position templates:

1. *Postures from 3 flags memorized and a goalpost currently seen.* A short term memory for the bearings on the last three flags seen is used. Estimated distances

to these landmarks and odometry are used to update theses memorized flags when the robot moves. Besides the memorized bearings on flags, also the angle to the goalposts when seen in the current frame is used. From all possible combinations of 3 flags, or 2 flags and one goal, a robot posture is determined by triangulation.

2. *Postures from current percepts.* For this, only the current seen percepts with reliable distance information are used. From all possible combination of 2 flags or one flag and one goal, a robot posture is determined. If a sample *s* has low probability *p*, it is replaced by a robot posture stored in the buffer of position templates. If there are not enough candidate postures in the buffer, low-quality samples are replaced by samples with random coordinates.

4.3. Behavior control

This chapter covers the behavior control of the current architecture. Behavior control can be seen as the brain, the upper command of the robot. Behavior control uses information about the world as input, and then gives commands (such as walk with speed x, look to direction y etc) to motion control, dependent on its state.
As we have already explained in section 3.2, the algorithm of the behavior control is not straightforward; it consists of many different behavior modules, and has many possible states. The current state of behavior control determines the way output is determined from input. In this chapter we will explain how behavior control works. We will explain about agent, options, and states and about how the robot changes between states.

The behavior control of the Dutch Team 2004 is written in XABSL, an XML based behavior description language; for more detailed information on the XABSL architecture we refer to The German Team Description [7].

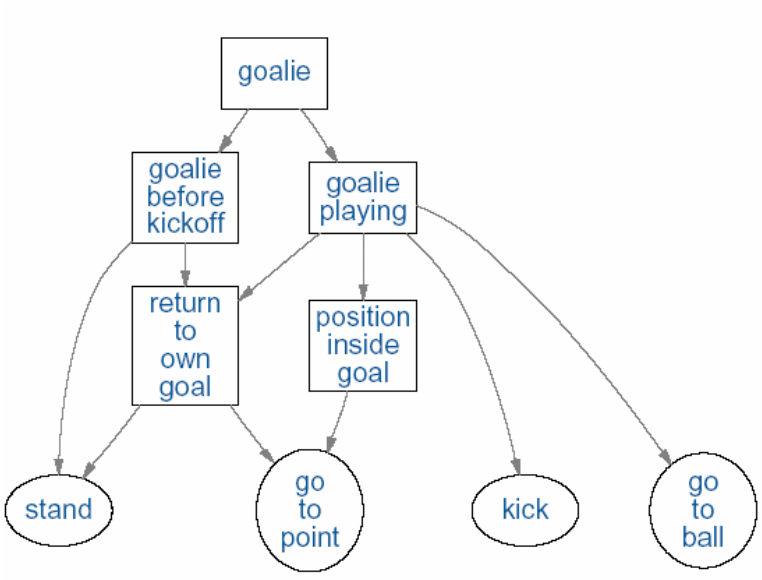


Figure 4.1. The option graph of a simple goalie behavior (from the German Team Report [7]). Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from an option.

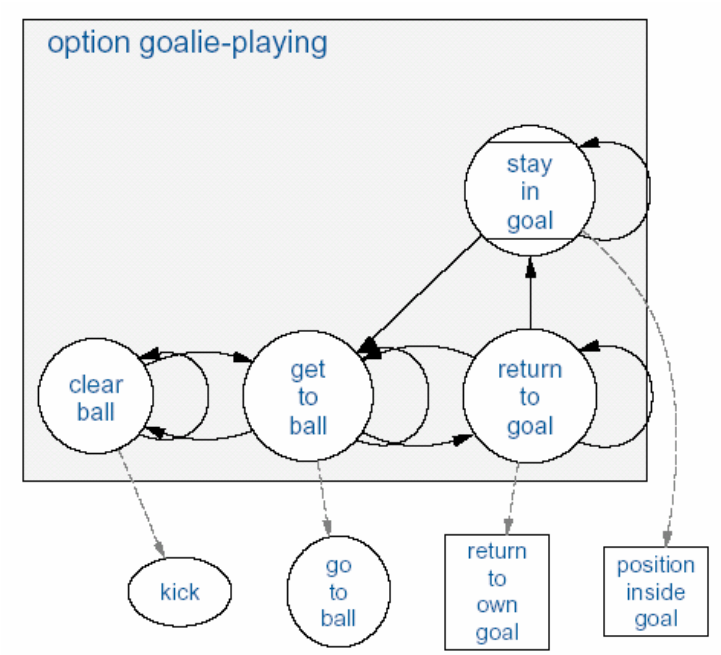


Figure 4.2. the internal state machine of the option "goalie-playing". Circles denote states; the circle with the two horizontal lines denotes the initial state. An edge between two states indicates that there is at least one transition from one state to the other. The dashed edges show which other option or basic behavior becomes activated when the corresponding state is active.

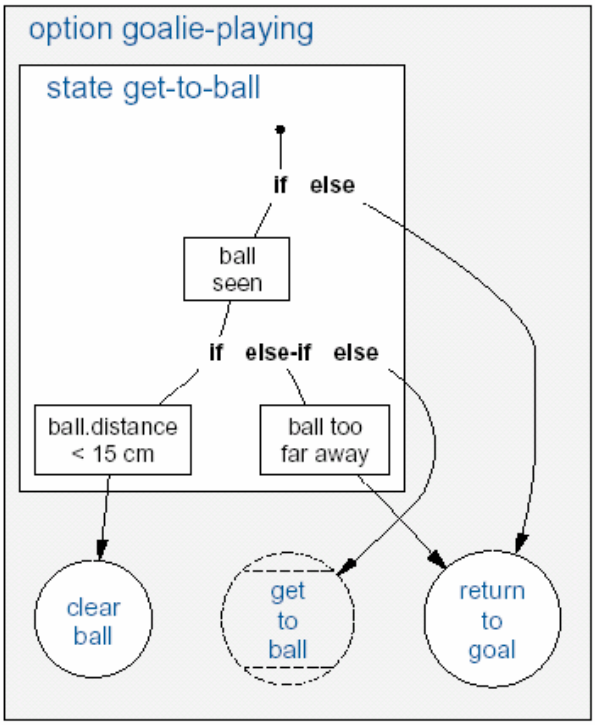


Figure 4.3. The decision tree of the state "get-to-ball". The leaves of the three are transitions to other states. The dashed circle denotes a transition to the own state.

4.3.1. Agents, options and states

Agents

An agent is a solution of the behavior control. Similar to solutions of other modules, such as image processing, the current agent is found in the file *modules.cfg*, or can be changed from the PC that is used to control the team over WLAN, through *Robot Control*. During an actual soccer match only one agent is active. An agent consists of a number of behavior modules called options.

Options
The options are the behavior modules. Every option is stored as a different XML file (e.g. *goalie.xml*, *striker.xml*, …). Options can activate other options or basic behaviors.
In figure 4.1 one can see the options of a simple goalie behavior in an option graph. The connections indicate possibilities of options entering next options. The terminal nodes of the graph are called basic behaviors. Within options, the activation of behaviors on lower levels is done by state machines. Each option can have a number of states, from which always one is active (the current state of the option). An example of an option is shown in figure 4.2.

States
At state is a loop that can change to another loop (state), under certain conditions. Each state has a subsequent option (e.g. position inside goal), or a subsequent basis behavior (e.g. stand), determining the motions being executed. Each option has an initial state. This state becomes active when the option was not active during the execution of the option graph. Additionally, states can be declared as target states. In the options above, it can be queried if the subsequent option reached such a target state. This helps to check if a behavior was successful. Each state has a *decision tree* (figure 4.3) with transitions to other states as their leaves.

Setting output symbols
Additionally, each state can set special requests (output symbols), that influence the information processing besides the actions that are generated from the basic behaviors. The output-symbols mainly just give a value to some parameter that is used as input for e.g. the motion control. An example of an output symbol is the kind of head-motion carried out. If in a certain state, the robot is supposed to look at the ball, it is set in the following way:

<set-output-symbol ref="head-control-mode" value="head-control-mode.search-for-ball"/>

4.3.2. The DT2004 behavior agent

In figure 4.4 you can see the first layers of the options of the DT2004-soccer agent, used for the RoboCup games in Lissabon. When a robot is in a certain state low in the hierarchy, it also is in all the states above. When for instance the robot is in penalized mode (dark blue options in figure 4.1-4.3), a total of 4 options is executed every timeframe (table 1).

All robots use the same agent
In figure 4.4, one can see that all robots play with one agent; player roles are determined lower in the behavior architecture. This is done mainly because the RoboCup rules require a set of common tasks from every robot. (Init/ set/ ready/ play/ penalized/ final). Thus when a robot is playing, it always first executes the subsequent options: play soccer module switch, play soccer, playing standard. Then it will execute the other options.

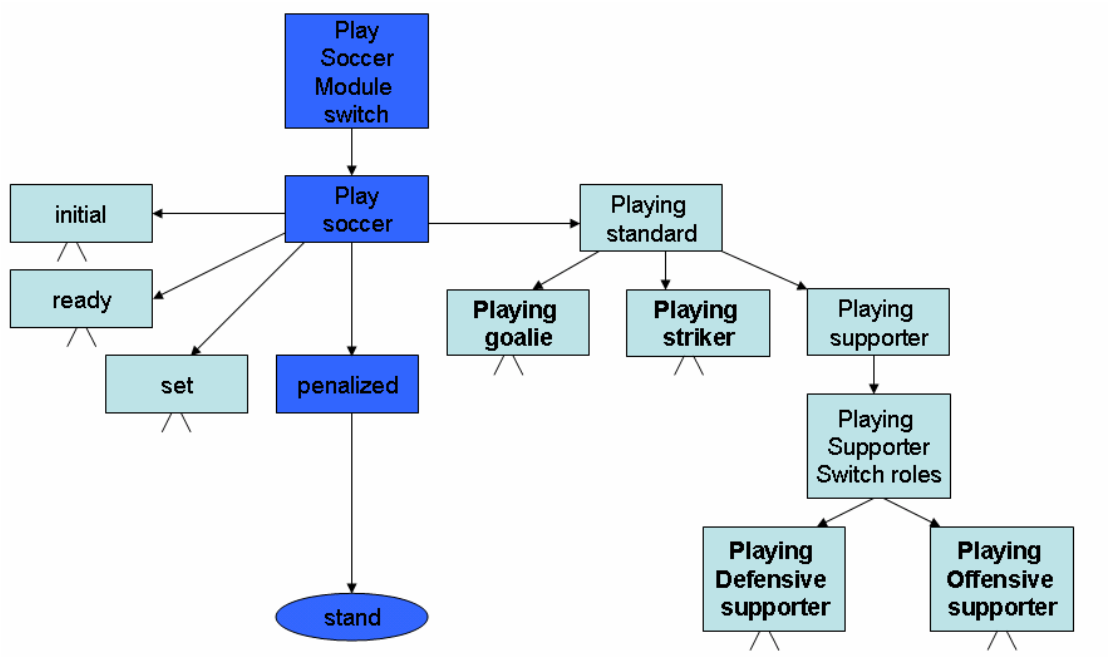


Figure 4.4. General simplified layout of the first layers of the behavior Architecture of the DT2004-soccer agent. The rectangular shapes indicate options; the circular shape indicates a basic behavior. When the robot is in penalized state, all the dark-blue options are active.

Table1. The states of all the options active when the robot is in penalized state.

Option	**State**
Play soccer module switch	Play soccer
Play soccer	Penalized
Penalized	Stand
Basic behavior: stand	

How players get roles

We have already seen that different robots with different task (goalie/striker) use an identical agent. The different roles are determined somewhat lower in the architecture. One can also see that there is no state for a defender and second attacker. A defender is actually a "defensive supporter" in the state "playing defensive supporter". Robots use the variable player-role in choosing their state. A robot can get a new role in 2 ways:

1. Initially, when a new game starts, the robot finds its role from the file *player.cfg*.
2. During the game, the *defender*, *striker1* and *striker2*, can dynamically switch roles. When the defensive supporter comes very near the ball, it changes its role to *striker2*, and sends a message to the other robots. If the current *striker2* gets this message, it will change its own role to defender.[1]

[1] Note that this is very sensitive to mistakes. In the RoboCup in Lissabon, where the WLAN was inactive during many games, it happened a lot that a player changed role, but its message didn't reach the other players. Thus we ended games with 1 goalie, 3 strikers, and no defensive and offensive supporters.

4.4. Motion control

Motion control is the part that calculates the joint-values of the robots joints. Three types of motion can be identified in the Dutch Aibo Team: *special actions*, *walking motion* and *head motion*.

Special actions
A *special action* is a set of joint-values that is executed sequentially. The designer can set such a set of parameters in a motion file (*.mof*). All kicking motions, get-up actions and the special movements executed when goals are scored are special actions. The *special actions* control both the leg joints and the head joints.

Walking engine
The *walking engine* is used for all walking motions. The engine takes a large set of parameters (approx. 20) that result in walking motions. The designer can change these parameters in order to get a different walking motion. A set of parameters can be stored and can be available to behavior-control. E.g. one set of parameters is used for fast walking; another set is used for turning with a ball. The *walking engine* mainly controls the leg joints.

Head motion
The head joints are controlled by *head control*, independently from the leg joints. The head motions are mainly just (combinations of) predefined loops of head joint values. The active head motion can be controlled by behavior control. E.g. there are solutions for the robot looking from left to right at the height of the landmarks (*searchLandmarks*), at the height of the lines (*searchLines*) or the robot can look at the ball (*searchForBall*).
There are also combinations available, such as *searchAuto*. In this mode the robot looks at the ball every few seconds, and scans the field otherwise.

5. Estimating the quality of self localization

Before we can discuss why using localization-and behavior-information in a behavior dependent architecture can lead to a better vision system, we need to understand what makes up a good vision system. In this chapter we will discuss the most important factors that determine the quality of the self localization. We will discuss what are the factors, how their performance depends on which parts of the system, and we will derive one formula for making a rough estimate of the quality of the system from the systems characteristics.

5.1. Main factor: quality of image processing

In a vision-based localization system, the most important part is, of course, the quality of the image processing. We will define the quality of image processing first by the number of true and false positives, and then with a correct acceptation rate (*CAR*) and a false acceptation rate *(FAR)*. Finally, we will show the causes of the errors made by the image processing system.

5.1.1. Measures for the quality of image processing

True and false positives

When starting a game or after being picked up by a referee, robots don't know where they are. Also do robots loose their position and orientation when performing a motion ($\Delta_{Odometry}$); therefore they need sensor information to update the knowledge of their position. The vision based autonomous soccer playing robots can relocate themselves by detecting known objects, such as goals, flags and field lines in the camera images. If an object is detected correctly by the image processing, this is called a true positive (Fig 5.1a). If an object is detected that is not really there, this is called a false positive (Fig 5.1b). True positives increase the quality of self localization, false positives decrease it.

a) b)

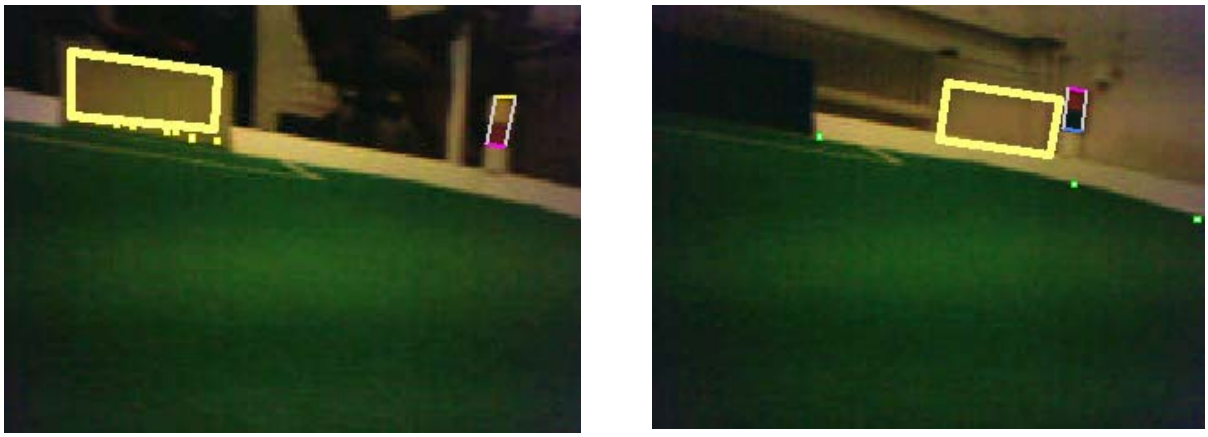


Figure 5.1. Images taken with an Aibo camera. a) Two true positives. Both the yellow goal and a yellow/pink flag are detected. b) One false positive: The Image Processor mistakes the wall with a yellow goal. The pink/blue flag is detected correctly.

Acceptation rates: CAR and FAR

For the quality of the self localization, the number of true and false detections within a certain time frame is relevant. Not only is it important that more true than false positives are detected, it is also important that true positives are detected at a rate that is high enough to correct for

the error made in the odometry while walking. Consequently, we will work with a Correct Acceptation rate (CAR) and False Acceptation Rate (FAR), defined as:

$$CAR = \frac{N\ True\ Positives}{image}\ ; \qquad CAR = \frac{N\ False\ Positives}{image} \tag{5.1}$$

N and $N_{visible}$

The quality of self localization depends on the total number of true and false positives, which is the sum of the contributions of many sub-algorithms (for goal, flags, etc). The total acceptation rates can be defined in the following way:

$$CAR_{tot} = CAR_{goal1} + CAR_{goal2} + CAR_{flag1} + ... = \sum_i CAR_i \tag{5.2}$$

$$FAR_{tot} = \sum_i FAR_i \tag{5.3}$$

The contributions of these individual algorithms, not only depend on the quality of the algorithms; they also depend on whether or not an object is visible in the image. The CAR of goal detection can only be non-zero if a goal is visible in the camera image. We will make the rough approximation that all individual algorithms for object-detection have equal impact and equal characteristics (5.4).

$$CAR_i = \begin{cases} CAR' & if\ object\ is\ visible \\ 0 & otherwise \end{cases} \qquad FAR_i = FAR' \tag{5.4}$$

Filling (5.4) in (5.2) and (5.3) leads to (5.5) in which $N_{visible}$ is the total number of objects visible in a camera image and N the total number of objects searched for by the image processing algorithms.

$$CAR_{tot} = N_{visible} * CAR' \qquad FAR_{tot} = N * FAR' \tag{5.5}$$

Note that we make a very rough simplification; individual algorithms with identical characteristics are very rare. However, our result (5.5) is qualitatively valid: If many objects are searched for relative to the number of objects visible in a camera image, the performance of the image processor will likely be low.

5.1.2. Reasons for low quality image processing

We have seen that good image processing means that many good and few false objects are detected. How often this happens is a result of the quality of the algorithms (CAR and FAR), on how many objects are visible in images ($N_{visible}$), and for how many different objects is searched (N). Below we will describe the main reasons why the image processing so often fails when the robots play soccer, i.e. what causes CAR to be low, FAR to be high and N to be relatively high compared to $N_{visible}$.

1. Influence of lighting conditions

When the temperature or intensity of the lighting chances, so do the corresponding values of Y, U, V of the image-pixels. If a color-table is calibrated for one lighting condition and subsequently the lighting changes, the colors in the image will be wrongly segmented. The algorithms that highly depend on color will start making mistakes; the CAR becomes low, the FAR high.

2. Overlapping color values
Pixels belonging to two different objects, who are supposed to have a different color, often have identical pixel hue values. It is not always possible to distinguish green from blue or sky-blue, yellow from white, pink from orange or red. If one calibrates a color-table in such a way that the whole of the blue goal would be detected as blue, many parts of the green carpet will be also be segmented as blue (Figure 5.3) and the FAR will become high.
3. Objects not visible
There are many situations in which the robot cannot see any object, for instance when he is handling the ball, is surrounded by players, or is near the border facing the audience. No matter how good the algorithms are, the CAR_{total} will still be zero, since $N_{visible}$ is zero. The image processing can only produce errors.
4. Using only few classifiers of objects
The main part of e.g. the goal image processing algorithms, is finding blue objects (or yellow for the yellow goal). Only some shape information is also used for the detection. This makes it very likely that other blue objects, such as the blue spots in the carpet in fig 5.4, are wrongly detected as goals.

Figure 5.2. Lighting changes from left to right: 1-TL-lamp, 2-TL-lamps, 2 - TL- and 1, 4 floodlights.

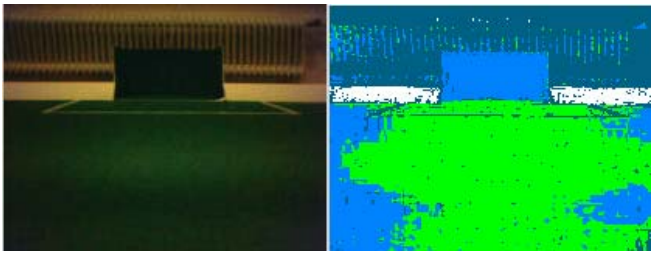


Figure 5.3. Overlapping colors. The color of many pixels of the green carpet is identical to the color of the blue goal. Picture from an Aibo camera, shutter set to fast

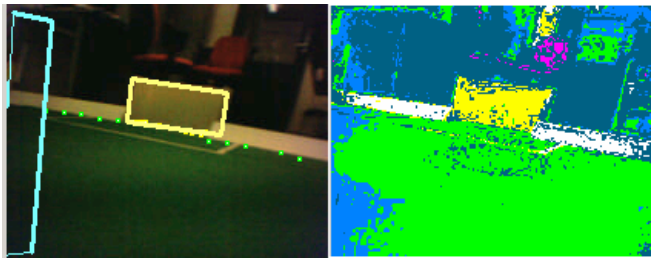


Figure 5.4. Few classifiers. A blue goal is falsely detected.

Note that 2 of the 3 factors that were mentioned (changing lighting and overlapping colors), are factors resulting from the fact that we are using color segmentation for classifications. These problems could disappear if one would move towards color independent algorithms (shape-based). Shape-based algorithms will of course bring there own problems due to changes in the perceived world.

5.2 Other factors driving performance

Not only does the quality of self localization of the robots depend on what the robot sees and how images are processed. We will describe a few factors that influence the quality of self localization and have no direct link with the processing of camera images.

5.2.1. Old quality

The quality of a known position at a certain time is highly dependent on the history of self localization. We will represent this dependence with the factor Q_{old}, the quality of self localization in the previous frame.

If a robot has just started in a certain position it knows that the chances are high that it will still know its position accurately a few seconds later, Q_{old} is high. If the robot is picked up and placed somewhere else on the field, its localization will be totally wrong and this will stay wrong until the robot uses the positions of objects that it detected in the world, for its localization; in this situation Q_{old} is low.

5.2.2. Error in odometry

When robots walk, there is a difference between where they think they walk and where they walk. This error we call $\Delta_{Odometry}(x,y,\theta)$ and this will contribute in a decrease of the quality of self localization. The size of the error depends both on the accuracy of the odometry information and on the kind of motion the robot executes: If the robot stands still, the error will be zero. If the robot walks straight, the error in the x and y direction will increase. If the robot turns or does a kicking motion, mainly the error in the θ-direction will increase.

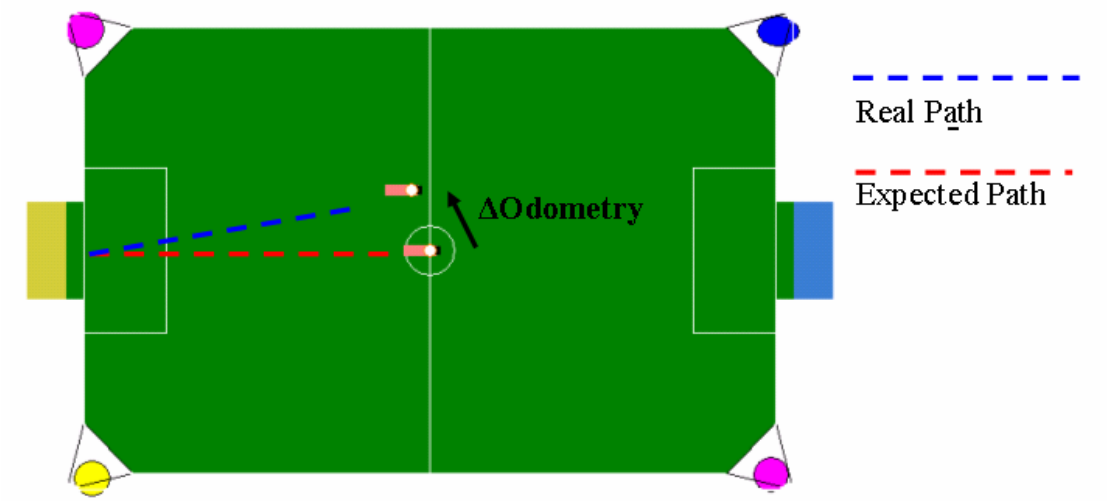


Figure 5.5. The robot is supposed to have walked from the yellow goal to the center of field, but in reality has done a slightly different motion; this results in that the robot knows less good where it is.

5.2.3. Update rate

For the quality of self localization it is not only important how many true and false positives are detected (CAR_{total} and FAR_{total}), it is also important how soon these detected percepts result in an updated robot pose (Fig 5.6). To be able to discuss this phenomenon, we will introduce a parameter, the *update rate* (*UR*), a qualitative parameter that indicates the impact of a single percept on the self localization. A low *UR* means that many percepts are needed before the samples are significantly changed. A high *UR* means that already a few new percepts can lead to a new robot pose. The highest possible *UR* is obtained when the robot pose is analytically calculated form sets of perceps. Note that the update rate is not a quantitative parameter: how percepts influence the self localization depends on the nature of the detected percept, the settings and algorithms of the self locator and the state of the samples, not on only a single parameter.

Robustness versus speed

A system with a low update rate will be highly robust against false percepts; a system with a high update rate will be able to relocate very fast when the robot is picked up by the referee or

when the robot moves around very fast and makes large errors in its odometry. Note that the optimal update rate highly depends on the situation. A robot standing still doesn't need to update very often, thus it can have a very low update rate to allow for robustness against false percepts. A robot walking around very fast, doing kicking motions and colliding with other players will make significant errors in its odometry. For this robot a system with a high update rate is better.

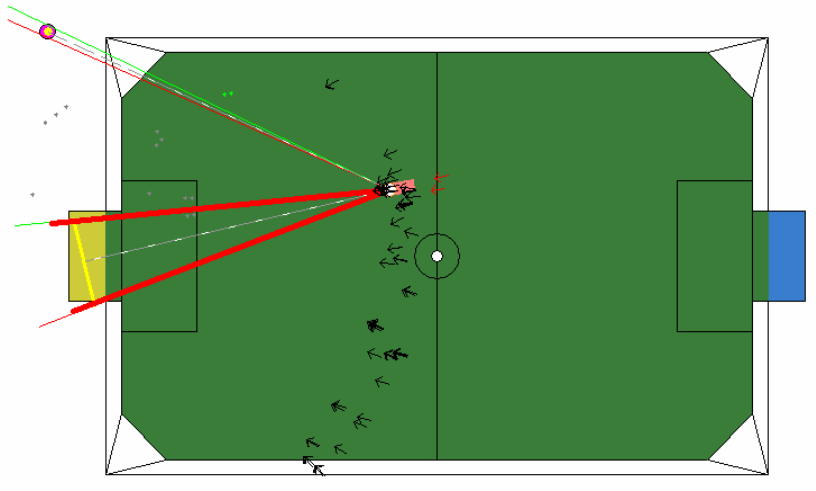


Figure 5.6 The robot locates itself by updating many samples with information from the detected percepts, in this case a yellow goal and pink/yellow flag. A high update rate means that only a few percepts will lead to a new position. A low update rate means that percepts only slightly influence the samples (black arrows).

5.3. Relation between quality and factors

Main formula for quality

We will summarize all factors driving the performance of self localization in one formula for evaluating the expected quality Q_{new} of the self localization after evaluation of a new image:

$$Q_{new}(x,y,\theta) = Q_{old} - \Delta_{odometry} + UR(x,y,\theta) * \left(CAR_{tot} - FAR_{tot}\right) \quad (5.6)$$

Filling (5.5) in (5.6.) results in:

$$Q_{new}(x,y,\theta) = Q_{old} - \Delta_{Odometry} + UR * (N_{visible} * CAR' - N * FAR') \quad (5.7)$$

Q_{old} is the quality of self localization after the previous frame. The quality increases with true positives (CAR_{tot}) and decreases with false positives (FAR_{tot}). The update rate (UR) determines their influence. $\Delta_{Odometry}$ represents the error made in odometry in one time-step. Note that the quality is evaluated in x, y- and θ-direction. When e.g. a robot has just been turning, the quality of its x- and y-position could be high, while the quality of θ would be low.

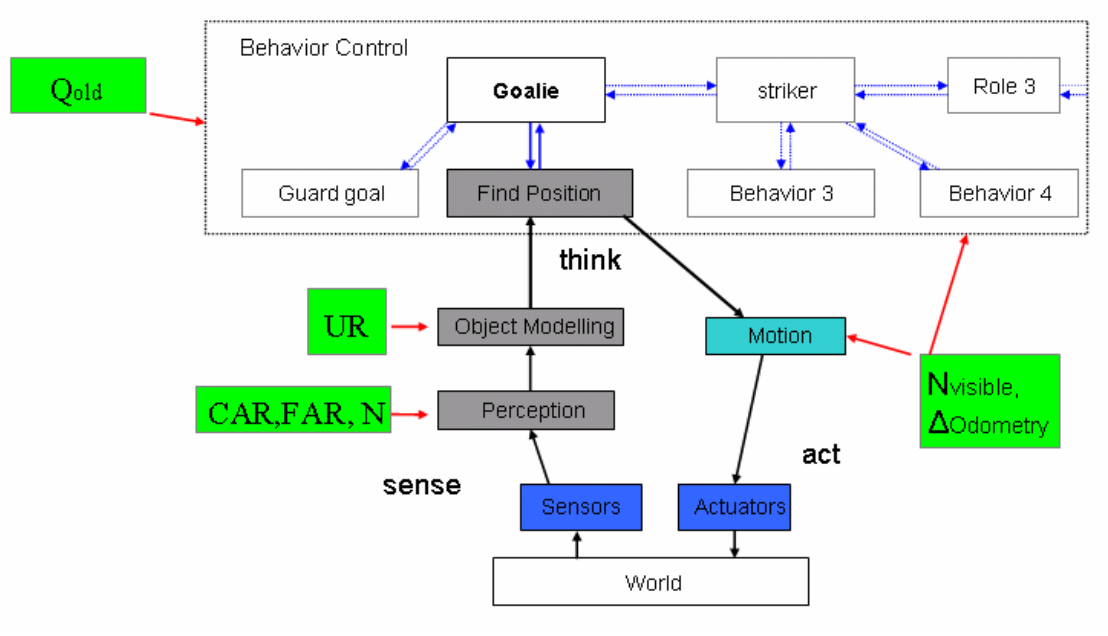


Figure 5.7. The robot architecture with the factors that determine the quality of its self localization.

Below is described how the factors are influenced by the different parts of the system.

CAR', FAR'	The correct and false acceptation rates of the individual algorithms depend only on the quality of the image processing algorithms running in the image processor that was used.
N	N is the total number of objects searched for in images by the image processing algorithm that was used.
UR	The update rate is a characteristic of the self localization algorithm.
Q_{old}	The old quality is correlated with the state of the behavior control. After catching and kicking a ball, the quality will be lower than when the robot has just been walking around searching for landmarks.
$N_{visible}$	The total number of objects visible in the camera image depends on the state of the robot (its body and head) which on its turn is highly dependent on what the robot wants to do (behavior). For a robot looking at a goal, $N_{visible}$ will be higher than for a robot looking in the audience. Note that when a robot holds the ball with its head, the camera is faced directly to the ground and $N_{visible}$ equals zero. In this case, every possible image processing can only lead to a decrease of the quality Q.
$\Delta Odometry$	The error in the odometry is directly coupled with the motion carried out. Indirectly, $\Delta_{Odometry}$ is coupled with the current behavior, since that determines what motion should be executed.

5.4. Summary

We have derived a qualitative formula for estimating the quality of self localization of a robot system:

$$Q_{new}(x,y,\theta) = Q_{old} - \Delta_{Odometry} + UR * (N_{visible} * CAR' - N * FAR') \quad (5.8)$$

The quality of the self localization depends on:

- The quality of the image processing algorithms (*CAR'* and *FAR'*)
- $N_{visible}$ in relation to N
- The Update Rate versus $\Delta_{Odometry}$

The quality of the self localization not only depends on the quality of the image processing algorithms, it also is highly behavior- and position dependent!

6. A behavior based vision system

In order to make a big step in improving the performance of the soccer-playing AIBOs, we have designed and implemented a behavior based vision system. We stepped away from the concept of using one general vision system for the entire robot system.
In behavior based vision, specific methods for image processing and self localization are used for each different behavior. This will improve several characteristics of the AIBO software:

- *Improved robustness of the self localization*
The robustness of the self localization is currently a major limitation of the performance of the robots. Under many circumstances, our robots do not know where they are and consequently they are not able to play proper soccer. We will show how using a modular Image Processor that uses location and behavior information can lead to a robot that localizes better and in a wider variety of circumstances (more robustness).

- *Increased available processing power.*
Another issue in the current robot systems is the limit on the available processing power. This in combination with the requirements of real-time processing limits the possibility of using advanced image processing algorithms in current robots. Using image processing algorithms only in certain specific situations, instead of at all times, can increase the effective available processing power and can allow for more advanced algorithms.

- *Reduced system complexity*
A major problem of the current system is that it has become far too complex. The behavior-tree and image processing algorithms have become so large and complex that it has become nearly impossible to oversee how changes in certain algorithms influence the entire system. By breaking apart (and hence simplifying) the behavior tree and connecting the image processing algorithms directly to the behaviors, we can achieve a system in which it is far more understandable what really happens.

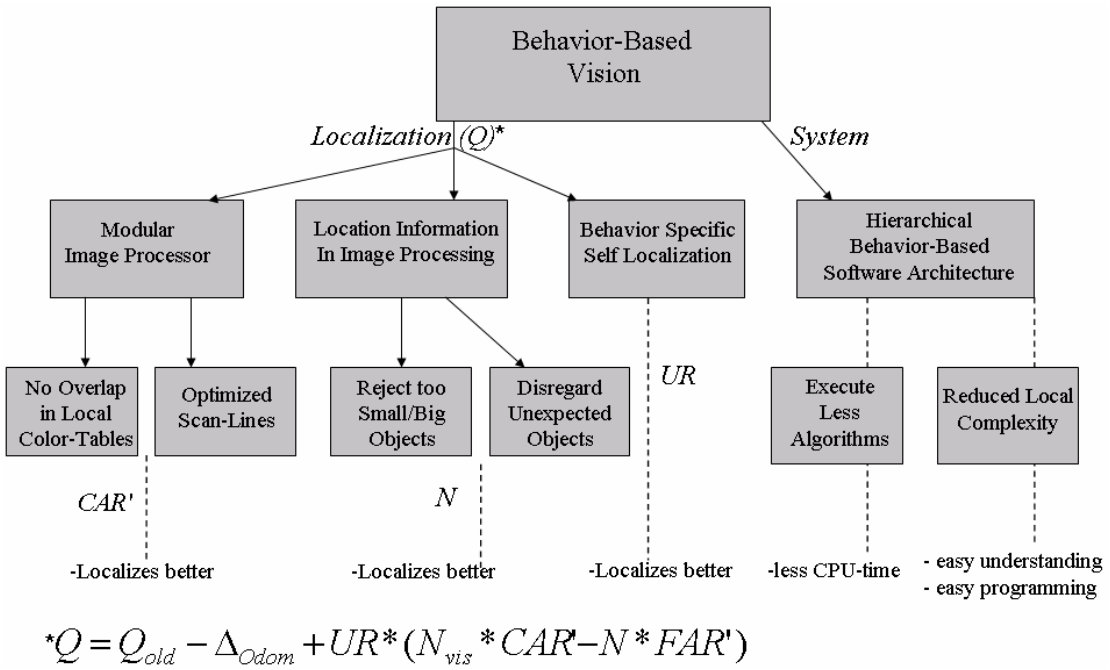


Figure 6.1. The different ways in which behavior based vision leads to a better robot-system with better self localization.

6.1. Modular image processing

We will use a modular image processor and use algorithms that are entirely object dependent. This means that not only will we use separate algorithms for e.g. flags and goals, but we will also distinguish between the detection of the blue and yellow goals and the blue and yellow flags.

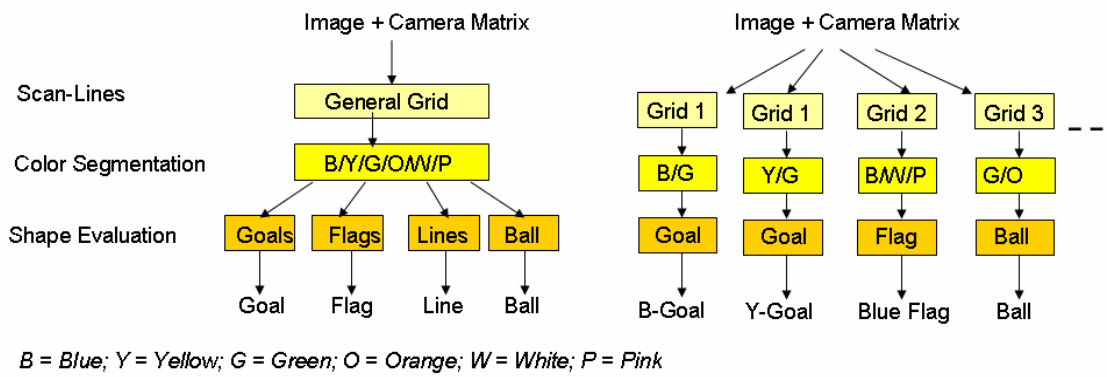


Figure 6.2: General versus modular image processing. Left one can see the general image processing. A single grid and color-table is used for detecting all candidates for all objects. In the modular image processing, the entire process of image processing is object dependent.

6.1.1. Object dependent color-tables

A blue/green pixel observed well below the horizon, will likely belong to the green field while a blue/green pixel seen at horizon level will likely belong to the blue goal. We can, for instance, use this fact by implementing algorithm dependent color tables. Instead of using one general color-table calibrated for all 10 colors, we use a larger number of different color-tables only calibrated for the colors relevant for a certain algorithm. For the detection of the yellow goal we only use a color table calibrated for yellow and green; for the detection of the blue flag we use a color-table only calibrated for pink, blue and white, etc. In figure 6.3 one can see how the same image is segmented using various algorithm specific color-tables.

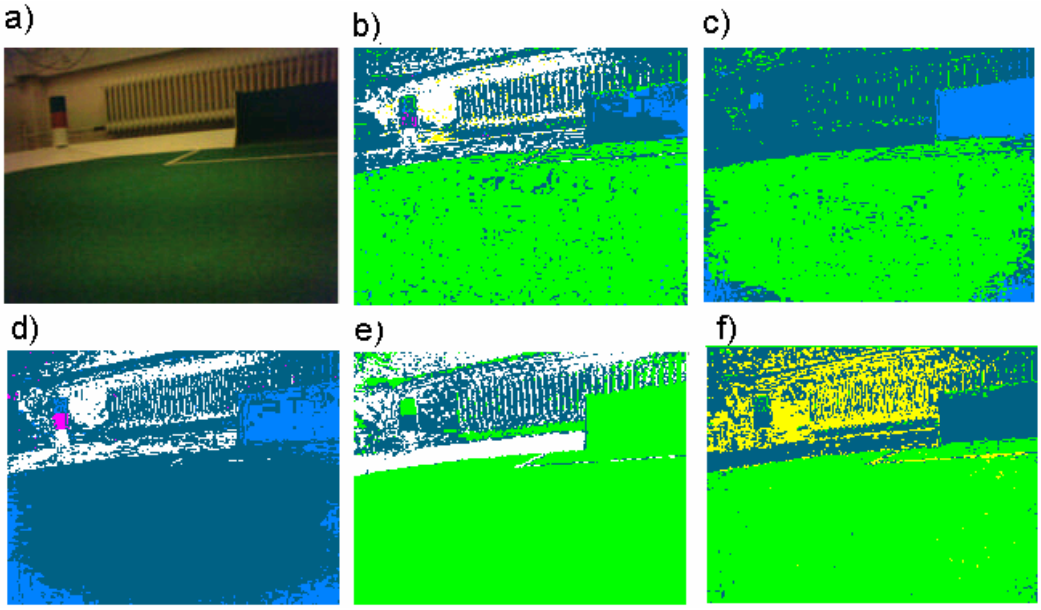


Fig 6.3: a) camera image; b) segmented with the general color-table; c) segmented with the blue/green color-table for the detection of the blue goal; d) segmented with the blue/white/pink color-table for the detection of the blue flag; e) segmented with the green/white color-table for the detection of the field lines; f) segmented with the yellow/green color-table for the detection of the yellow goal.

Advantages
Calibrating several independent algorithm-dependent color-tables each containing only 2 or 3 colors, greatly reduces the problem of overlap. Labeling many pixel values as one color for one algorithm will not affect the quality of another algorithm. Therefore the color spaces can be larger filled (in fig 6.3e, large parts of the color-space are defined as green). This has the following great advantages:

- *Calibrating color-tables is easier.* Calibrating a general color-table as in fig 6.3b is a tedious work and can take a very long time (over half an hour). One has to make sure that the overlap of 10 different colors is sufficiently small and one must test the color-tables on many algorithms in many situations. Calibrating a specific color-table as in fig 6.3e can be done in a few seconds; calibrating the total of 5 specific color-tables can be done in a few minutes.
- *Stricter algorithms are possible.* With the now largely filled color-spaces, greater parts of objects can expected to be segmented. (Note the difference of the goal in 6.3b and 6.3c). Thus the algorithms for shape evaluation can be made much stricter and hence the *FAR* can be reduced.
- *Variable lighting conditions.* When the color-tables are allowed to be calibrated more widely, they can also be calibrated to detect colors in a wider range of lighting conditions.

Using image dependent color-tables is a smart step from color dependence to shape dependence, without having to implement new algorithms or requiring more processing power. From algorithms dependent on a 10-color color-table we go to algorithms depending on a 3-color color-table. From here it is only a small step to algorithms with 0-color color-tables, that is, to completely shape-based algorithms. More about can be read in 11.2 (recommendations)

6.1.2. Local scanning dependent on the camera view

A flag is always visible at eye-level (around the horizon); the goals are somewhat lower in the image; field-lines and the ball are only visible well under the horizon. Therefore we can sustain with only scanning small parts of images for individual objects as can be seen in figure 6.4. Local scanning both saves processing power and reduces the chance on false positives.

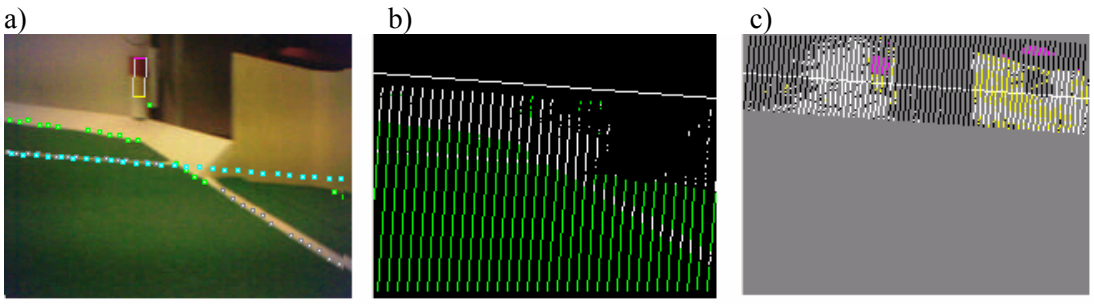


Figure 6.4: a) An image in which 2 lines and a flag are detected. b) Scanning for line-points is only done below the horizon. c) Scanning for flags is done in an area around the horizon.

6.2. Using location information in image processing

6.2.1. Disregard unexpected objects (reduce *N*)

The objects that are or are not visible in the robot's camera, depends totally on the robots state (the robot's pose and head pose). By only searching for visible objects (e.g. not searching for

the yellow goal when facing the blue goal), the performance can be increased: N can be decreased, while $N_{visible}$ is left untouched. The total false acceptation rate can be significantly reduced, without changing any algorithm (the CAR_i and FAR_i remain intact).

6.2.2. Using distance information

If the robot is far away from an object, it will appear small (fig 6.5a). When the robot is near it will appear big (fig 6.5b). An image processor that is using position information can reject candidates that are bigger or smaller than it would expect to see. Hence the false acceptation rates (FAR_i) of the algorithms are reduced.

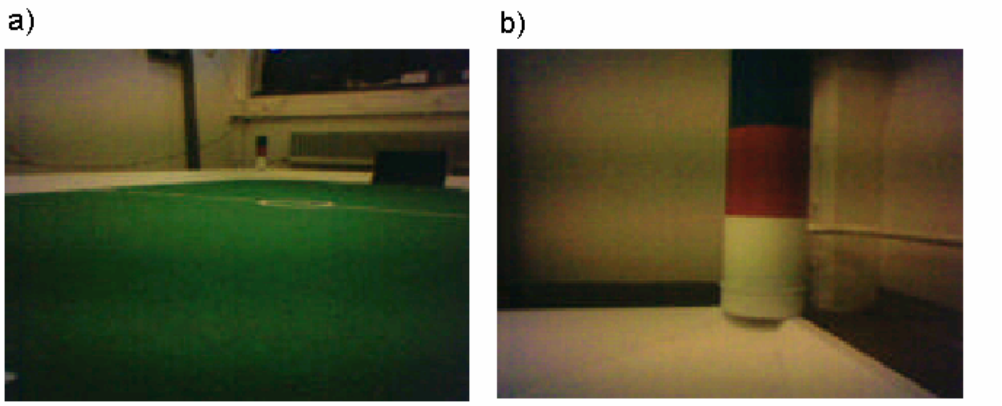


Figure 6.5.Two images of a blue/pink flag; a) at 5 meter distance; b) at 30 cm distance.

6.2.3 Danger of using location information

Using location information for optimizing the quality of self localization can lead to serious problems if the location is not evaluated correctly! If the robot faces the blue goal while thinking he faces the yellow goal and only tries to detect the yellow goal, he will detect no objects, and will never retrieve its correct position. Using prior knowledge can lead to wrong loops: the robot not knowing where it is can lead to a robot that is disregarding good information and will cause him to stay wrongly localized. Hence, when using location information, one needs a checking mechanism that can detect these wrong loops. E.g. one can check if enough expected percepts are detected. (Evaluate the false negative rate).

6.3 Behavior specific self localization

Performance increase can also be achieved by directly using position- and behavior information in the self localization algorithms.

6.3.1. Use location-dependent particle filtering

We can use location information in the particle filtering [11], [12], of the self locator. If we know where the robot was previously located and we know he hasn't moved significantly (Q_{old} is high), we can reject changes to samples indicating a totally different position (e.g. somewhere in the opponent half). In this way we can directly filter out some contributions of false positives, without analytically analyzing their nature. A visualization of this location-dependent particle filtering can be seen in figure 6.6. The self locator in 6.6b assumes that the robot is located somewhere in the vicinity of the yellow goal. Samples indicating otherwise are rejected.

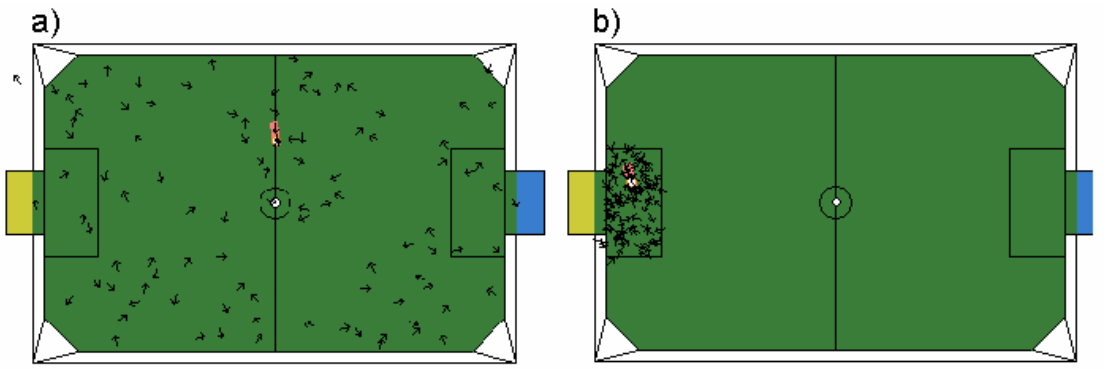


Fig 6.6. Using location-dependent particle filtering. a) The position is unknown, thus samples divided over the whole of the playing field are accepted; b) The player is known to be located somewhere in the yellow goal-area, samples indicating a different position are rejected.

Note 1: this location-dependent particle filtering mainly will work much better for translation than for rotation information. Because of collisions, while turning or doing kicks, a large error in the rotation position can occur very easily. Unnoticeably walking from one half to the other takes much more time.

Note 2: using location-dependent particle filtering will introduce the same dangers as only using percepts that are expected to be seen: the algorithm can possibly enter a local loop. If the robot actually stands in the middle of the field but thinks he stands in its own goal, he will just disregard all percept information and will continue to be located wrongly[2]. Mechanisms for detecting local loops are required. Another possibility is to use location-dependence in the particle filtering only for short periods of time, e.g. only in the behaviors where the robot is handling the ball.

6.3.2. Using behavior information for setting the update rate

If a robot doesn't know where he was and walks around searching for landmarks, the image processing will likely give high quality output. Using a high update rate is logical. If a robot has just correctly located himself and will be active in e.g. walking to while looking at a ball, the quality of the output of the image processing will be low. Using too much of this information for self localization will do more harm than good. Using a low update rate is best.

6.3.3. Adapting to the requirements of behaviors

Until now we have only shown techniques that will lead to an optimization of the quality of estimation of the robot's pose in absolute coordinates (x, y θ). For some behaviors however, the knowledge of these absolute coordinates is not so relevant, but the accuracy of some relative parameter is. Often it can be a much easier problem to robustly determine this relative parameter, than indirectly deriving it from the known absolute robot's pose. *By way of example:* for a striker who has the ball and wants to turn to the goal and kick, the only really important parameter is the angle between himself and the (free angle of) the goal. Solely the accuracy of this parameter will determine whether the kick will be successful (figure 6.7). In the current software, kick-angles such as the ones in figure 6.6 are derived from the absolute coordinates (x, y, θ) of the robot. In order to obtain an accurate angle, both x, y, and θ need to be evaluated accurately, requiring many different objects (goals, flags, lines) to be seen many times. An erratic detection of one of these objects results in a worthless kicking motion. However, one can also directly derive such an angle directly from a detected goal. For an accurate turn-and-kick motion, only the position of the goal will have to be determined correctly once. This is a much easier problem and hence the system can be designed far more robustly.

[2] Often a human vice too.

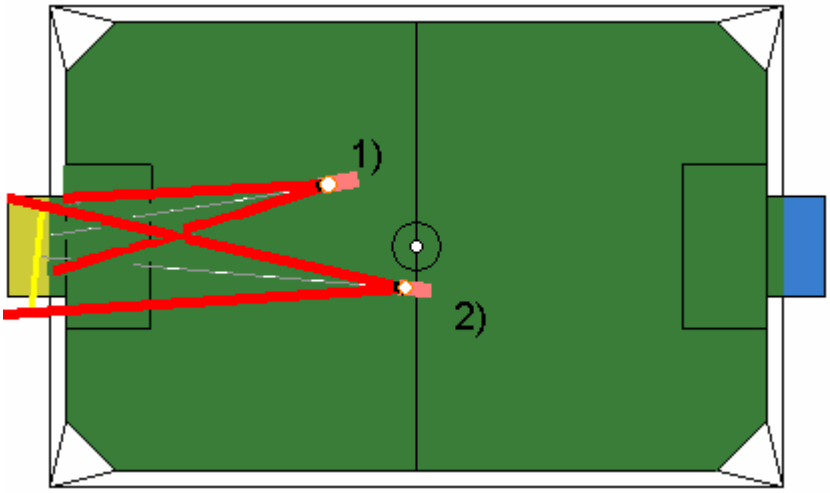


Figure 6.7. Angles between the robot and the yellow goal-posts. For the striker who wants to kick a ball to the goal, thinking his own position to be 1) results in an equal performance as thinking he is in position 2). The accuracy of the absolute x, y and θ is irrelevant.

6.3.4. Multiple self locators versus a single self locator with parameters

Behavior specific algorithms for self localization can be implemented by actually building multiple self locators, or by using a single self-locator that can use different methods for updating samples (table 6.1).

Table 6.1. The differences between multiple self locators and one self locator with parameters

Multiple self locators	**One self locator with parameters**
Different sample-sets	One sample-set
Information in samples lost when switching solutions	Information preserved when switching solutions
Different files	One file
Entire method can be different	Only the update process of samples is different
Individual self locators are less complex	The overall system is less complex
Easier to develop self locators	More reusability

What is most logical for implementation depends not only on how similar or different two desired self localization algorithms really are. It depends also on how often the robot is expected to switch between solutions. Monte Carlo based self localization methods have important history information stored in their particles and require many image frames to be evaluated before the pose evaluation stabilizes. If a highly frequent switching between solutions is expected, it is not advisable to use multiple self locators.

In chapters 7 and 8 both types of implementations are used for the goalie with behavior based vision. An entire new self locator for localizing on lines is used for the goalie. The sample-set in this self locator, however, is used both in the situation where the goalie stands in its goal (update rate high, no particle filtering) and for the situation where the goalie clears the ball (update rate low, particle filtering). The general self locator is used, with specific parameters, for the situation that the goalie needs to return to his goal.

6.4 Hierarchical behavior based software architecture

In the previous chapters we have shown in which way the quality of self localization can be improved by using position- and behavior information. In this chapter we will show how to

implement position and behavior information in the software. We present a behavior based architecture. The current algorithms used for image processing and self localization are directly coupled with the behavior that is executed.

6.4.1. Why not to use a more general purpose vision system

In the current software, the robot operates as a single sense-think act loop and the system uses a more or less general purpose vision system. The algorithms for detecting objects in images and updating the robot‘s pose from perceived percepts are always the same. The main advantage of using a general vision system is that the reusability of the code is 100%. All the code is executed every frame. Thus only one set of algorithms has to be programmed for the whole system.

The main disadvantages of the general vision system are its complexity and the processing power it requires. All algorithms are executed and thus they require the same processing power every frame. To totally understand the system in one situation, one has to understand all the vision algorithms.

It is theoretically possible to implement the use of prior knowledge (and thus obtain the advantages described in chapters 6.1-6.3 on top of the existing system, and thus maintain this general vision system (as can be seen in figure 6.8) We can use distance information by not just looking at e.g. flags, but by looking both at flags near and flags far. One can use the color information by looking not just at goals, but by searching both for the yellow and blue goal. We could also build a filter for rejecting detected objects that are not likely to be seen (rejecting the blue goal while facing the yellow one). Theoretically the self localization could be built in a way that it uses odometry information. It could use a high update rate if the robot is walking and a low one if the robot is standing still. Although theoretically it might be possible to build a system as in figure 6.8, this is not what we want to do. Besides the fact that the required processing power will increase (it might be doubled), the system will become far too complex. The system in 6.8 will require years for building and understanding. In a situation in which master's students contribute to the vision parts of the robot software a behavior based system is probably preferable over the design of a general vision system.

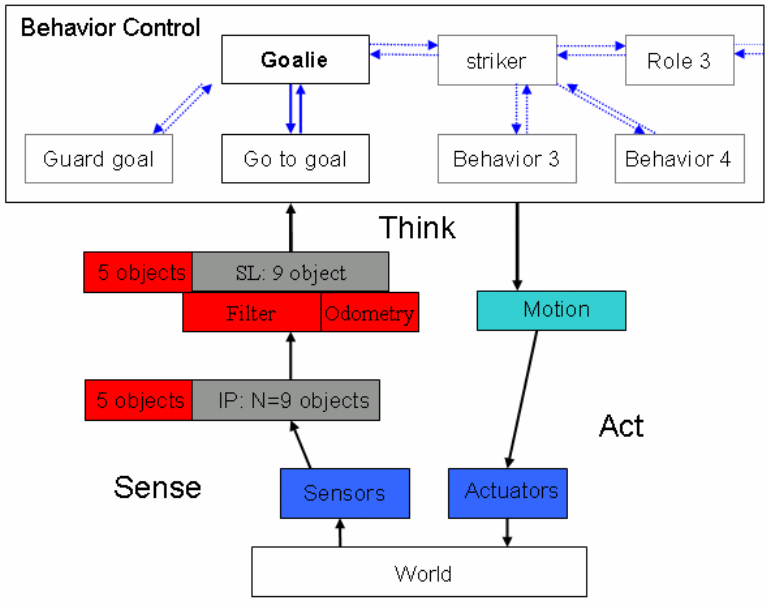
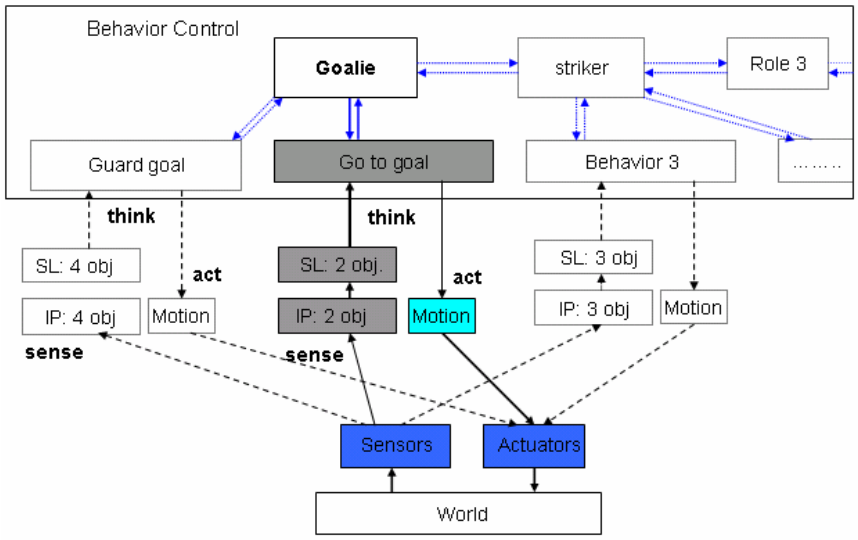


Figure 6.8. Using location- and behavior information in a general vision system. Lots of complexity and new algorithms are added on top of the exiting system that is one single sense-think-act loop.

6.4.2. A behavior dependent architecture

In figure 6.8 we have seen that using position and behavior information in the old system with one sense-think-act loop is practically not feasible mainly because of the complexity.
Instead we will implement a software architecture with a behavior dependent vision system.



6.9. Behavior based software architecture. Many different sense-think-act loops are possible. The current behavior determines which algorithms are used for image processing and self localization.

In the behavior-based software-architecture the sense-think-act loop will be split into many smaller loops. Each of these loops could operate as a totally autonomous system. Only one of these sense-think-act loops is active at one time. The current behavior determines which this is.

The main difference between the behavior dependent architecture and the old architecture is that algorithms are not always executed. If for a *go-to-goal* behavior the robot needs only to detect its own goal, the algorithms for detecting the opponent goal, the flags, the ball, the players, etc. are not executed and not used for self localization.

The increase in performance is due to the following two aspects:

- *Processing power*. Algorithms are not always executed. If in one behavior only the algorithms for the detection of 2 or 3 objects are executed of a total of 15 different algorithms, this greatly reduces the required processing power for image processing.

- *Complexity*. If for one behavior only a few algorithms are executed and the self locator only works on a few possible objects, this greatly reduces the local complexity. For instance in a *goalie return* behavior, making use only of the detection of the own goal for localization, debugging would be very easy. If the behavior doesn't work properly (the goalie doesn't return to goal), the problem can only be in the algorithms for own-goal detection. The designer doesn't have to evaluate the algorithms for ball-detection, player-detection, flag-detection, etc.

7. Behavior based vision software architecture

This chapter describes the implementation of the behavior-based vision system. It describes how the choice of algorithms in the image processing and self localization is controlled in the behavior control. This chapter will also describe in what ways the necessary algorithms are structured in files.

7.1 Interface between behavior control and vision

The way behavior control communicates with the self locator and image processor is made analogue to the way behavior control communicates with motion control about what type of motion should be executed. The states of behavior control can set the parameter *task-vision-request* indicating what solution for the vision system should be used. An example is shown below:

```
<state name="return-goal">
        <subsequent-option ref="goalie-return"/>
        <set-output-symbol ref="task-vision-request" value="task-vision.return"/>
        <decision-tree>
                <if>
                        <condition description="finished">
                                <subsequent-option-reached-target-state/>
                        </condition>
                        <transition-to-state ref="stand"/>
                </if>
                <else>
                        <transition-to-state ref="return-goal"/>
                </else>
        </decision-tree>
</state>
```

Figure 7.1. Code from the file playing-goalie-taks.xml; In the state goalie-return, the goalie is returning to goal, and the vision system uses solution "task-vision.return".

The output string *task-vision-request* is of the type *Enum*; the value *task-vision.return* is of the type number. (0,1,2….). Currently there are 6 different values for *task-vision-request* implemented:

DT2004=0, POSITION, CLEAR, RETURN, ODOMETRY, STRIKER=5;

Note that the behavior states only control one parameter for the entire vision system: the solutions for image processing and self localization are not controlled separately.

7.2. The task image processor

The main control of the image processing can be found in the file T*askImageProcessor.cpp*. In this task dependent image processor, the main algorithm consists only of a switch statement. Dependent on the value of *task-vision-request*, a set of algorithms is executed (figure 7.2).
For the *task-vision-request = DT2004*, the old general image processor (*DT2004ImageProcessor.cpp*) is executed with the general color-table *coltable.c64*.

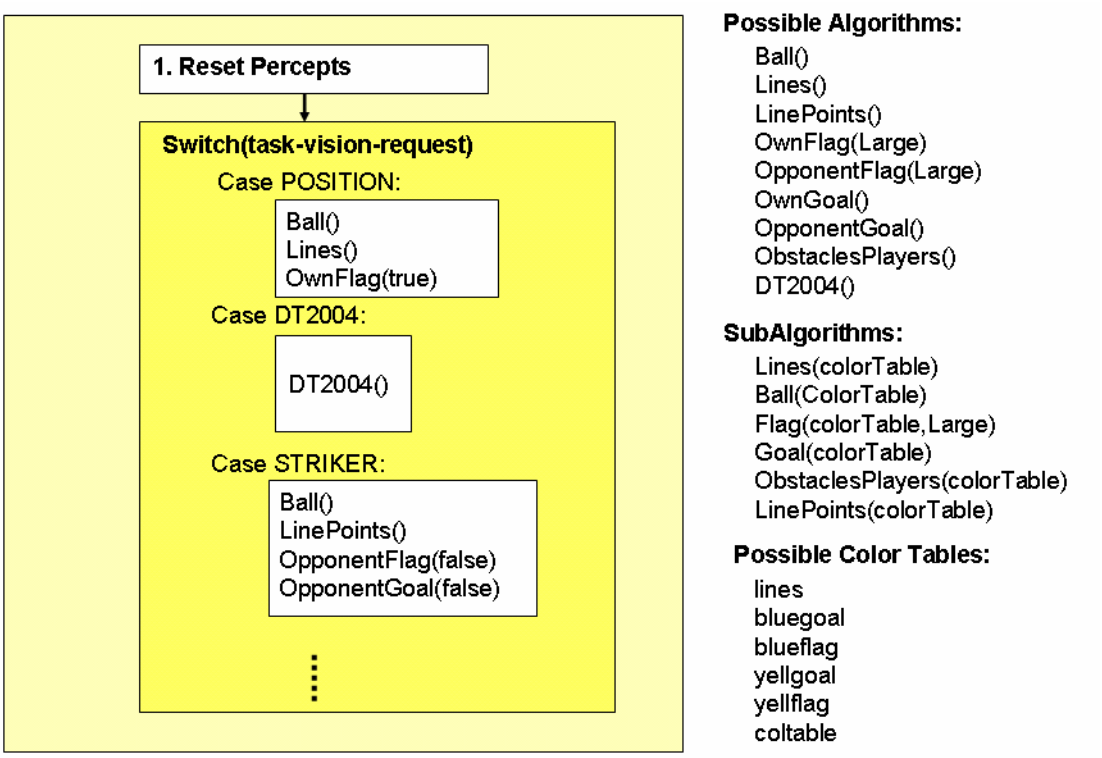


Figure 7.2 Main algorithm of the task-image-processor. For every frame, first the data-arrays of the percepts are reset. The next step is a switch statement. Dependent on the value of task-vision-request, a set of image processing algorithms is executed. All algorithms take the input image and a reference to a color-table as input.

Algorithms and sub-algorithms

In order to have a system that works for both team-colors (both for the blue- and red team), we have identified algorithms and sub-algorithms. Sub-algorithms take a reference to a color-table as input, normal algorithms don't. When e.g. the algorithm *OwnGoal()* is executed, first the own team color is evaluated. If the robot plays as *blue*, the sub-algorithm *Goal()* is executed with a reference to the color-table *bluegoal.c64*.

Color-tables

When the task image processor is initialized, 5 specific color-tables are loaded (*lines.c64*, *blueblag.c64*, *bluegoal.c64*, *yellgoal.c64*, *yellflag.c64*) in 5 color-table buffers (In addition to the already defined buffer for the general color-table *coltable.c64*).
The specific color-table-buffers are defined in *TaskImageProcessor.h*; the general color-table-buffer is defined in *Cognition.h*. The specific color-tables only contain a limited set of colors. E.g. the color-table *lines.c64* is only calibrated for orange, green and white, and is only calibrated for the detection of lines and the ball. Each sub-algorithm takes a reference to one of these 6 color-tables as input.

Files

Whereas in the old general image processor, the detection of all objects had its source in one file, (*DT2004ImageProcessor.cpp*), we have divided the algorithms for the sub algorithms in separate files. The files relevant for task dependent image processing can be found in figure 7.3.

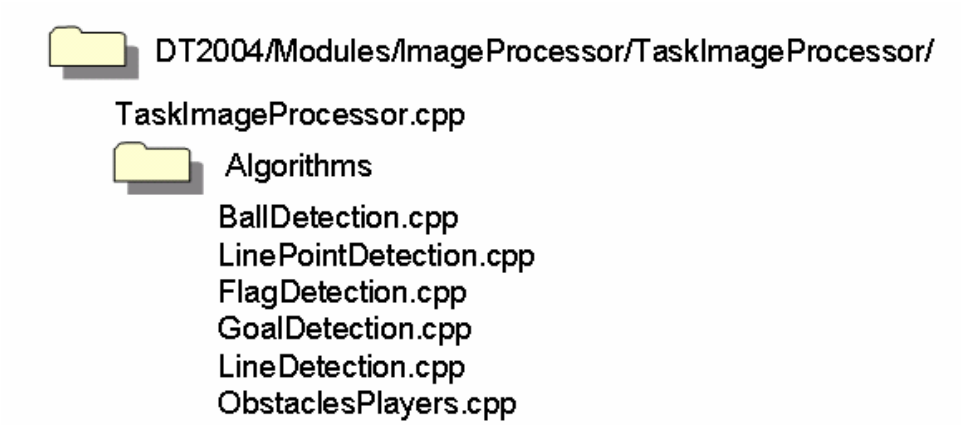


Figure 7.3. Files of the Task Image Processor. TaskImageProcessor.cpp resets the percepts and contains the switch statement from figure 7.2. The real algorithms for detecting balls, lines, flags etc are stored in the folder /Algorithms.

7.3. The task self locator

The task self locator works analogue to the task image processor. The core of the self locator (*TaskSelfLocator.cpp*) consists of a switch statement. Dependent on the value of *task-vision-request*, one of the self locator algorithms is executed (Fig 7.4).

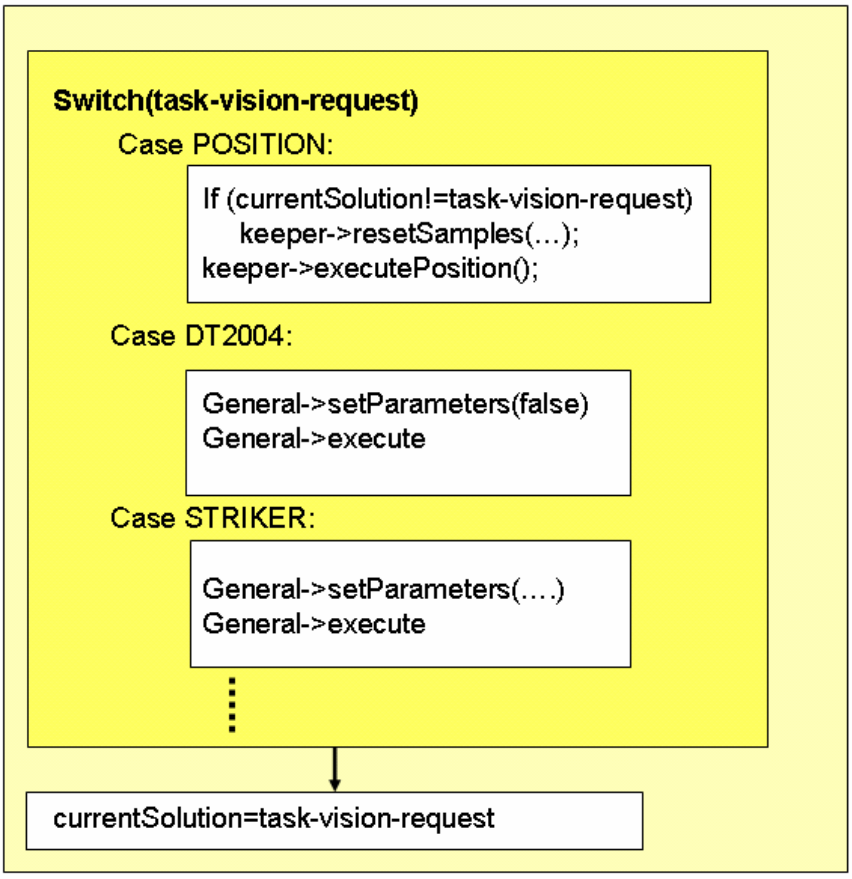


Figure 7.4. The algorithm in TaskSelfLocator.cpp. Dependent on the task-vision-request, either a sample-set or parameters can be set and a self-locator is executed.

General and specific self localization.
Dependent on the value of *task-vision-request*, one of the algorithms for the self localization is executed. We identify two types of algorithms: general and specific.

The general self locator is the Monte-Carlo based method as described in 4.2; it creates a robot-pose from all available percepts (flags, goals, line-points). The idea is that the general self locator is used for most of the behaviors.

Then there are specific self locators, designed only for very specific tasks. E.g., the goalie standing in its goal doesn't use the general self locator for self-localization, but localizes by directly obtaining the samples from perceived combinations of goal-lines. When the robot locates on odometry, it uses the *odometrySelfLocator* for localization.

Reset samples
Since we make use of more than one self locator, we also are working with several sets of samples. If because of a behavior change, the task-vision-request and a different self locator will become active, also a different set of samples will be used and the information in the previous set of samples will be lost. With *reset samples* we can reset the values of the samples of the newly active self locator, or we can transfer information about the robot's pose from one self locator to another.

Set parameters for general self localization
We have defined some parameters for the general self locator. These parameters define the characteristics of the general self locator. E.g., the influence of found percepts (goals, flags, …) in the update process of the samples is variable. Also it can be set whether or not the distance evaluation of a detected goal must be used when updating samples. With the value *FALSE*, the general self locator has exactly the same characteristics as the DT2004 self locator.

Files
The files relevant for self localization can be found in figure 7.5.

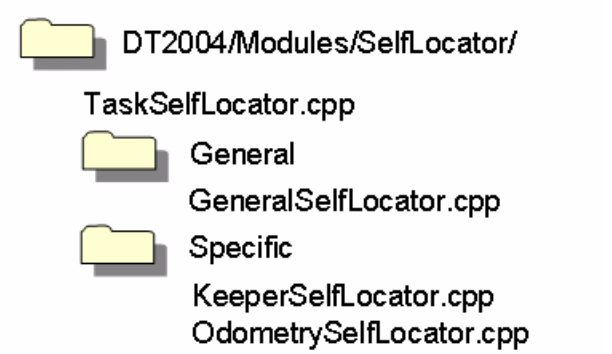


Figure 7.5. Files for the task self localization. The switch statement of figure 7.4. can be found in TaskSelfLocator.cpp. The general self locator is in the folder General. All other algorithms will come in the folder Specific. At this point only 2 specific algorithms are used. One for the goalie guarding the goal, and one for locating on odometry.

7.4. Settings in *Modules.cfg* for using task vision.
The file *Modules.cfg* requires the following values for using the task vision:

ImageProcessor TaskImage
SelfLocator TaskSelfLocator

Per default, *task-vision-request* is *DT2004* and thus per default, the *DT2004ImageProcessor* and *DT2004SelfLocator* algorithms will be used. The specific algorithms for image processing and self localization will become active when *task-vision-request* gets a value (in behavior control) other than *DT2004*.

8. Goalie-specific behavior based algorithms

This chapter describes the algorithms we have developed (or re-fabricated from existing code) especially for the goalie. These algorithms are implemented in the behavior based vision system as described in chapter 7.
In chapter 9 we will compare the performance of the goalie described in this chapter with a goalie using the DT2004 (general) vision system. We will use this comparison for evaluating the impact of using location information in a behavior based vision system.

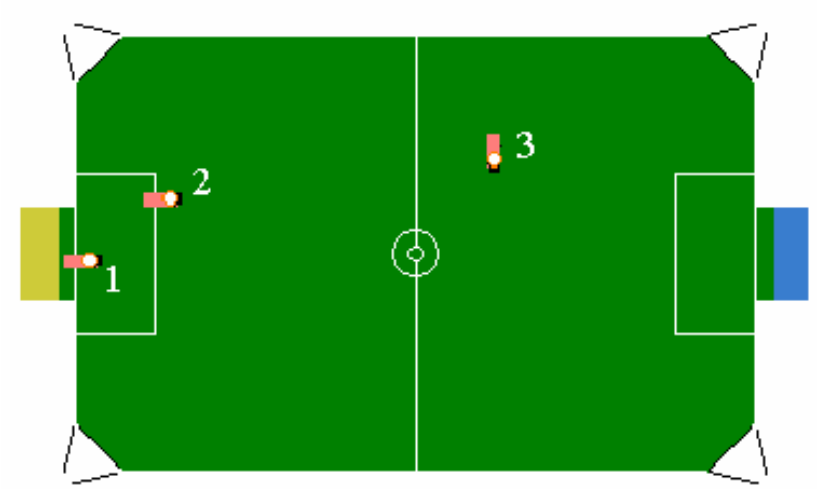
8.1. General overview of the goalie

We have roughly identified three situations for the goalie:

1. The goalie positions itself in its goal, knows where he is and the ball is not near.
2. The ball is near the goal area and the goalie tries to clear the ball.
3. The goalie has to return to its own goal.

In situations 1 and 2, the goalie mainly localizes on the field-lines surrounding the goal and on the two flags nearest to its own goal. In situation 3, when the robot walks back to its own goal, the goalie mainly localizes on its own goal.

a)



b)

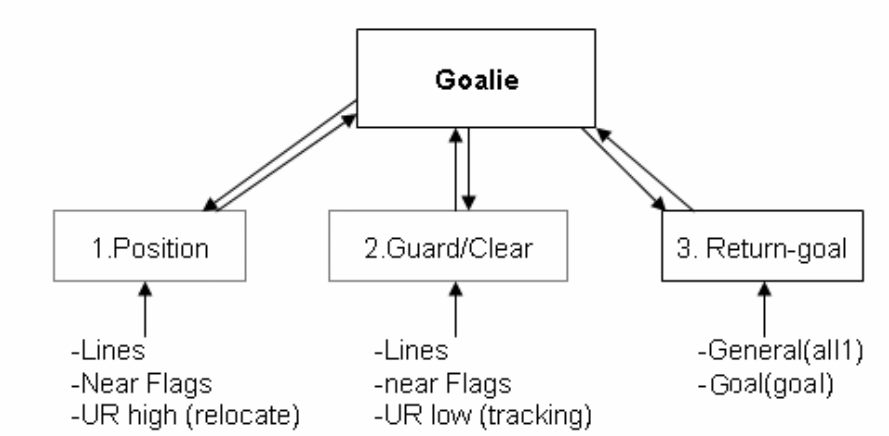
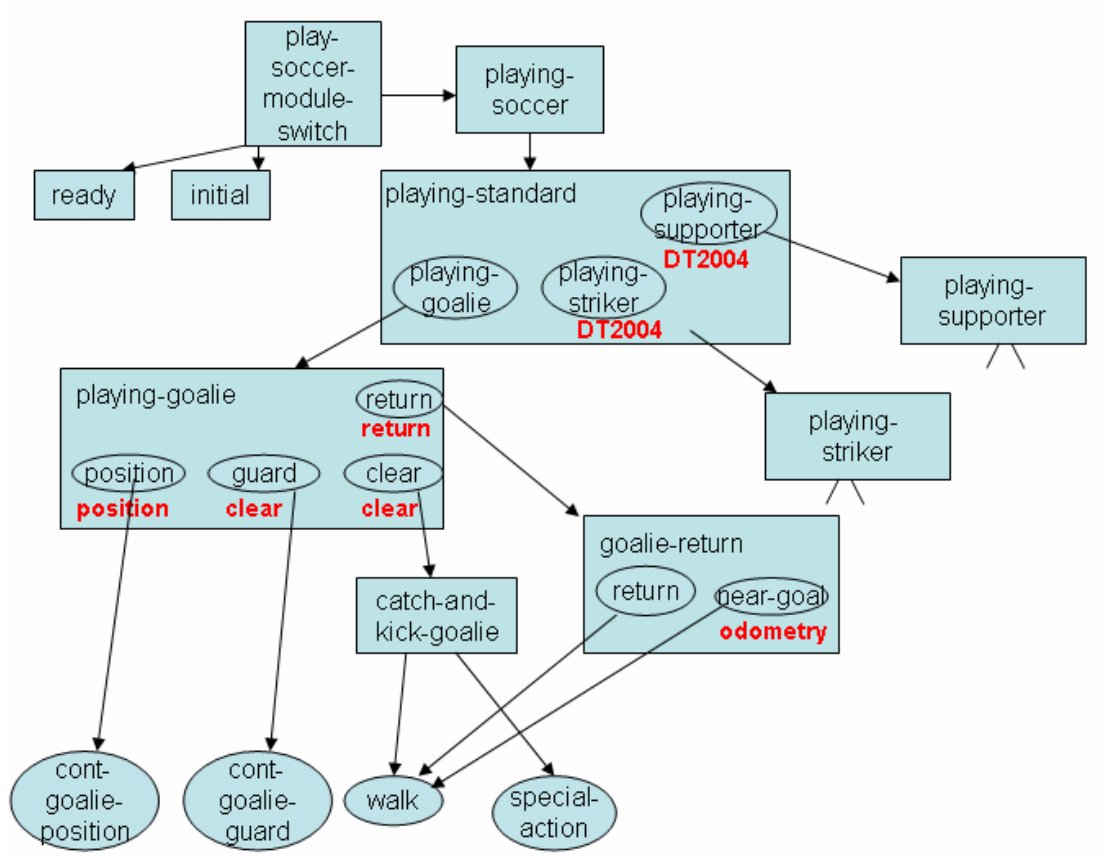


Figure 8.1. Schematic representation of a goalie. Three situations are identified: 1. the goalie stands correctly in its goal. 2. The goalie handles the ball in its own goal-area. 3. The goalie is far away from its goal and/or doesn't know where he is.

8.2. Behavior control

In figure 8.2 one can see the parts of behavior control relevant for the soccer-players with behaviour-based vision.



8.2. The most important parts of the soccer behaviour tree with implemented behaviour-based vision for the goalie. The squares indicate the different options. The circles in the squares indicate states. The value of task-vision-request belonging to a state is indicated in red. If the player is not the goalie, he plays with the DT2004 vision system. The goalie standing in its goal uses either the position or clear vision algorithms. In the return-goal behaviour, either the return or the odometry vision algorithms are used.

The red names in (8.2) indicate the *task-vision-requests* of certain behaviours. *Task-vision-requests* can be set anywhere in the behaviour trees. One can see that in the *playing-standard* option, the *task-vision-request = DT2004* for both the *striker* and the *playing-supporter* (*defender*, *attacker1*). Thus the general vision system is used for all players other than the keeper.

Playing-goalie

The *playing-goalie* behaviour (*playing-goalie-task.xml*) is the core of the goalie behaviour. When there is no ball, the goalie is in the state *position* with *task-vision-request = position*: the robot positions itself in the centre of the goal.

When the ball comes nearer the robot goes into the state *guard* with *task-vision-request = clear*. The robot will position itself between the goal and the ball.

If the ball is inside the penalty area, the robot goes into the state *clear*, still with *task-vision-request = clear*. The robot walks to the ball, catches it, turns away from the goal and kicks the ball (*catch-and-kick-goalie*). Afterwards the robot will go back to state *position*.

When in state *position*, the robot checks if its position is still correct, when it is not (he is not in its goal), it will switch to state *return-goal* with *task-vision-request = return*; the robot will walk back to its own penalty area.

Goalie-return
In the option *goalie-return*, the goalie first locates the own goal, then walks straight towards it; the robot turns when the angle to the own goal becomes too large. During the whole process of finding the goal, walking straight or turning, the robot uses *task-vision-request = return*, and the robot's vision system mainly works on the own goal. The last half meter of returning to the own goal, the goalie uses *task-vision-request = odometry*. This is because very near the own goal the error in the distance evaluation from the detection of the goal is too large.

8.3. Image processing

Below we will describe which image- processing algorithms are run at the different *task-vision-request*s. Also we will describe a few algorithms that we have changed especially for the goalie: flag detection for near flags, line-detection (vs. line-point detection), and goal-detection.

8.3.1. Algorithms for various task-vision-requests

Position and clear
When the goalie is near the own penalty area, he localizes mainly on the field-lines and the own flags. Also the ball has to be detected. The following algorithms are active both with the *task-vision-request*s *guard* and *clear*:

Lines(Hough=true)
OwnFlag(LargeFlag=true)
Ball()

Return
When the goalie returns to its own goal he localizes mainly on its own goal. Since the own goal is not visible from all positions on the field and absolute coordinates are not retrievable from one goal only, also a few flags and line-points are used for self localization.
The following algorithms are active with *task-vision-request = return*:

Lines(Hough=false)
OwnFlag(LargeFlag=false)
OpponentGoal()
OwnGoal()

Note that the robot does not try to detect the ball when returning to its goal. Also the robot doesn't perform the Hough transform for detecting lines from line-points.

8.3.2. Own flag Detection
For the *own flag detection*, we use mainly the algorithms written by the German Team [7], based on the scan-lines and the additional shape evaluation. Only in the shape evaluation we do some things different.

- *Calculate the size of the flag from both the yellow and the pink parts*. In the DT2004 algorithm (8.3b), the size of the flag was determined only from the yellow part of the

flag only; for increasing robustness, we have determined the size from both the pink and the yellow part (8.3c).

- *Only allow for a minimum size of the flag.* When searching for large flags (*ownflag*(*TRUE*)) we reject all candidate flags which are too small in width and/or height. Also too large flags, (probably erratic measurements) are rejected.

a) b) c)

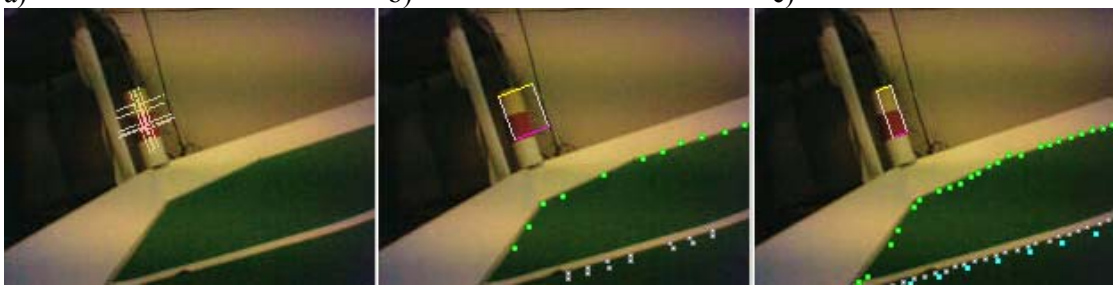


Figure 8.3. Flag detection a) Additional scanning on a few scan-lines, after a candidate flag was found; b) in the DT2004 algorithms, the width of the flag is determined only from the yellow part. Mistaking white wall-pixels for yellow flag-pixels can lead to a detected flag that is too large; c) in the OwnFlag() algorithms, the width of the flag is determined from the smallest width (pink or yellow part), leading to a more robust (and smaller) flag-detection.

8.3.3. Line detection

The main difference between the *lines(TRUE)* and the *lines(FALSE)* algorithm is that the *lines(TRUE)* algorithm detects 2-dimensional lines, with an angle θ and at a distance R relative to the robot. The *lines(FALSE)* algorithm only detects 1-dimensional points belonging to the lines, with coordinates (x,y,0).
The *lines(TRUE)* algorithm uses a Hough-transform to retrieve a line L from a set of line-points. The algorithm for detecting the line-points is identical to the one used in the *LinePoints()* algorithm (used in the DT2004 software).

Detecting line-points
For detecting the field-line points, we use the same scan-line algorithms that were already available from the German Team [7]. Green-white-green transitions indicated field-line-points; white-white-green transitions indicated border-points. In Figure 9.4 one can see some field-line and border points in camera images.

Fig 8.4. Images from the robot's camera when standing in the goal and performing a search-lines head motion. The white dots indicate the field-line points, the green dot, indicate border-line points.

Using the Hough transform for detecting lines.
We use a Hough transformation to determine at what line (R, θ) the points (x, y) are located. The line-points and the line are related in the following way:

$$R = x\cos\theta + y\sin\theta \qquad (8.1)$$

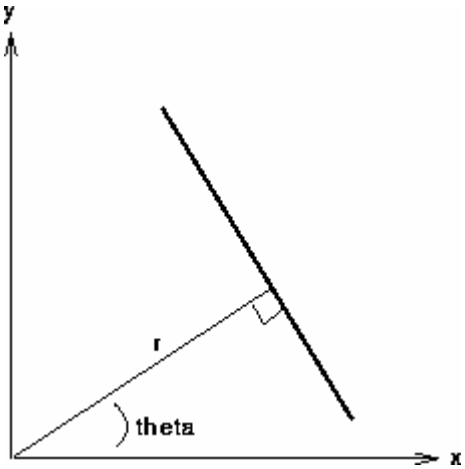


Figure 8.5. Representation of a line (R, θ) in (x,y)-space.

We find lines from points by letting all line-points contribute to one *HoughArray[R, θ]* in the following way:

```
For every point with coordinates (x,y)
    For all θ between -pi and pi
        R=x*cos(θ)+y*sin(θ));
            if(R>2 && R<159){
                HoughArray[θ,R]++
```

We can then find the lines simply by looking for the maxima in the *HoughArray[]*, the *θ* and *R* at which the *HoughArray* has its maximum are the coordinates of the detected lines. Two lines are found by first detecting the first maximum in the *HoughArray*, and then searching for the next local maximum at an angle of approximately *pi/2* (perpendicular).

A representation of the detection of lines can be seen in figure 8.6. The contributions of the line-points in *Hough-Space* can be seen in figure 8.7.

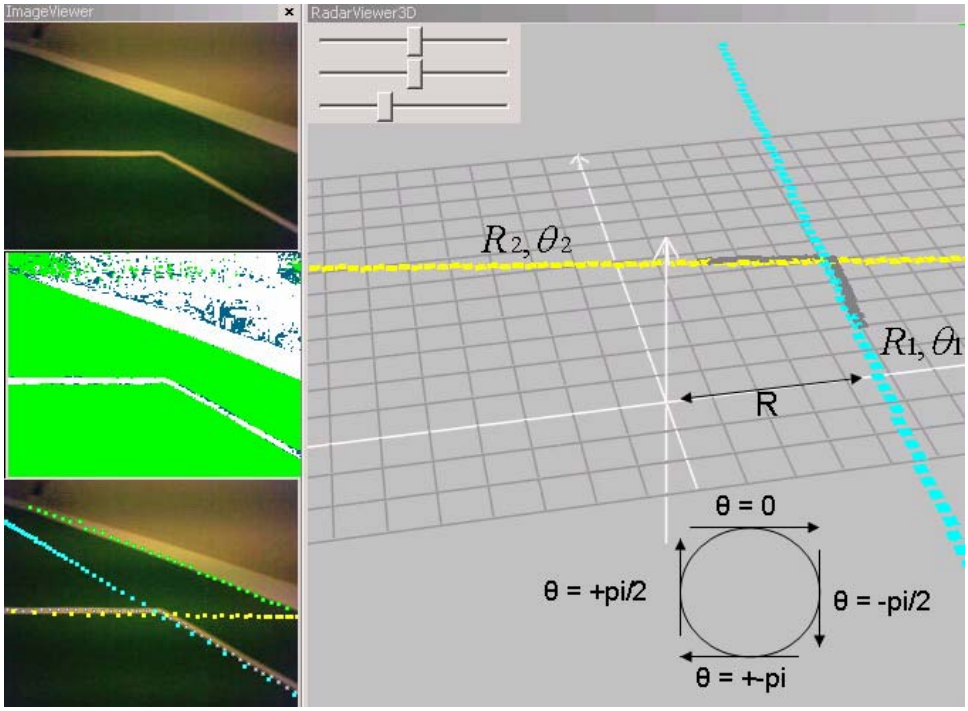


Figure 8.6. Detection of lines. Two lines (represented in blue and yellow) are evaluated from a set of detected line-points (grey points in the right image).

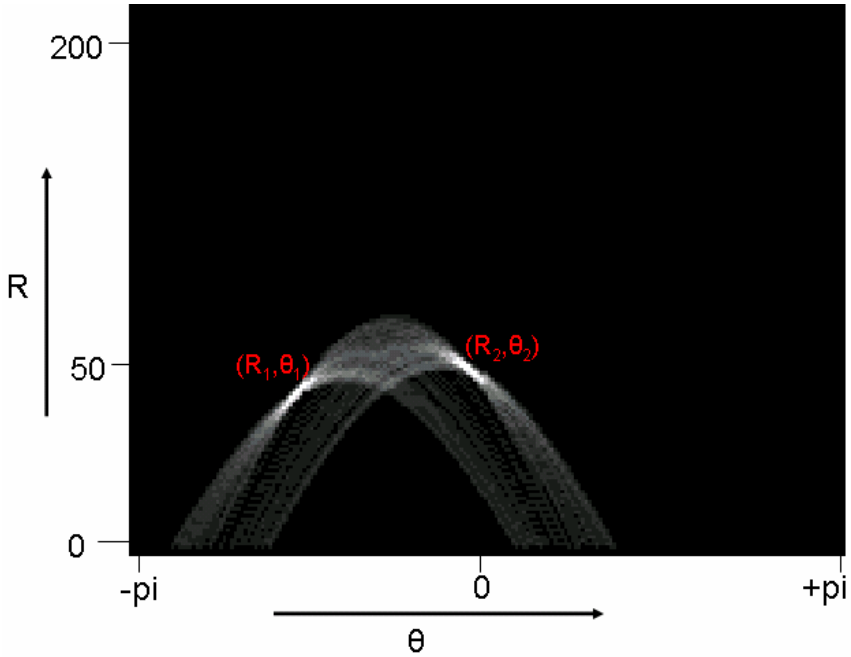
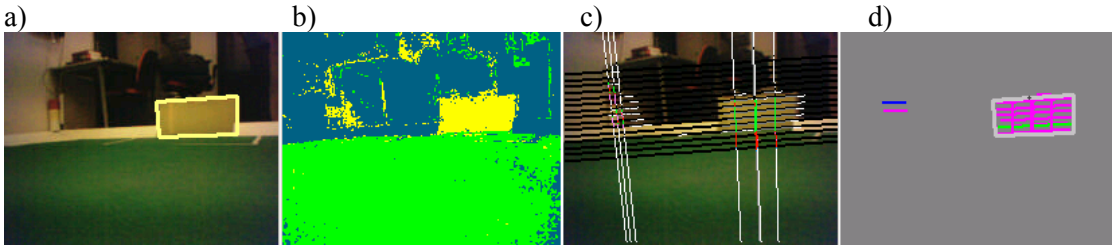


Figure 8.7. Contributions of the line-points of figure 8.6 in Hough-Space. The two lines are found at the whitest spots.

8.3.4. Goal detection

For goal detection we have used the algorithm that is almost identical to the algorithm that was used in the GT2003 software [7]. Goals are detected by scanning around the horizon in both horizontal and vertical direction for color (e.g. yellow) parts.
We have only changed a few things in the evaluation part of the scan-lines. We only accept candidate-goals if the yellow goal-points are followed (in vertical direction) by green field-points. This eliminates the chance of goals being detected in white walls.
Also we reject candidates if too many non-yellow pixels are detected within a yellow goal-part. The increased strictness of the selection is possible since the wide-filled yellow/green color-table (8.8b) results in far more candidate goals.

Some results of the goal-detection algorithm on a camera image can be found in figure 8.8.



8.8. Goal detection: a) raw image with drawn percept of goal; b) yellgoal.c64 *color-table; c) scan-lines used for scanning goal; d) results after scanning.*

8.4. Self localization

The keeper standing in its goal uses the keeper self locator (*KeeperSelfLocator.cpp*), a self locator depending on the field-lines and flags.
The keeper standing outside its goal uses the general self locator (*GeneralSelfLocator.cpp*), although with specific parameters.

8.4.1. Main algorithm of the keeper self locator

In figure 8.9 one can see the main algorithm of the keeper self locator. The algorithm uses many samples. They are updated by the odometry, by the lines, then by the flags; finally the samples are re-sampled. Every time-step the reliability is checked: it is evaluated if the keeper still stands in its goal. The algorithms for updating by odometry and re-sampling are identical to the ones written by the German Team [7].

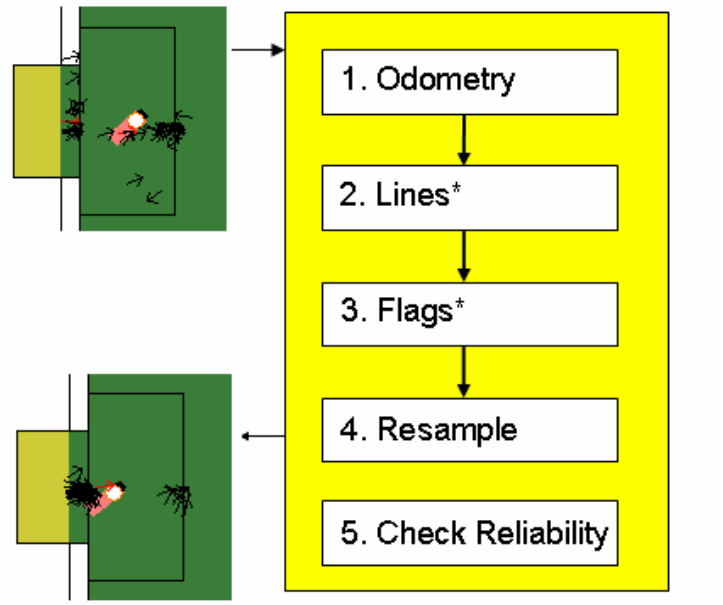
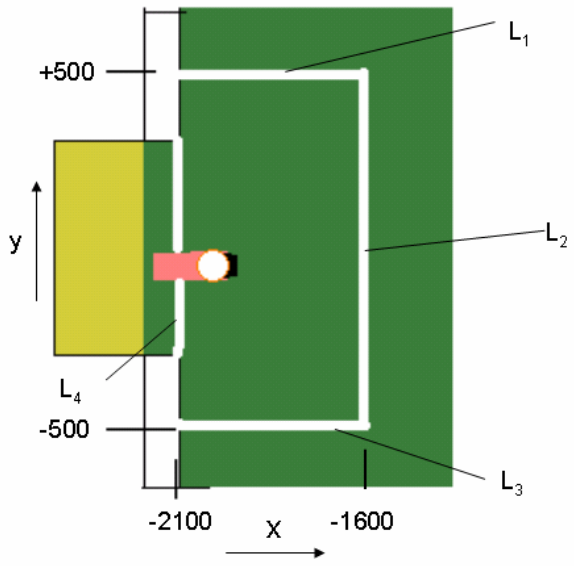


Figure 8.9: main algorithm of the keeper self locator. The (*) indicates that the steps are not identical for task-vision-request = position and task-vision-request = clear.

8.4.2. Lines



We want to retrieve our robot-pose (x, y, θ) from a detected line L (R, θ'). If we have detected a line and know which line we have detected, retrieving information from it is straightforward:

Line L_1:
$\theta = 1.57 - \theta'$
y = 500-R

Line L_2:
$\theta = - \theta'$
x = -1600-R

For Line L_3:
$\theta = -1.57 - \theta'$

y=-500+R *Figure 8.10. Penalty area*

For Line L_4:
θ =-3.14- θ'
x=-2100+R

However, we don't know if a detected line is L_1, L_2, L_3 or L_4 and thus we use tracking. For every sample it is determined what line is most likely to be seen:

For all samples S
Decide what Line L is seen
Update Sample S from L

The decision process evaluated the newly calculated pose (x,y, θ) for all 4 detected lines. The line resulting in a new position nearest to the old pose is chosen as the correct line. The update step moves the samples towards the calculated position. Also the quality is adjusted.

The difference in *position* and *clear* in line detection
The line update process with *task-vision-request = position* is slightly different from the one used for *clear*. For *position* we accept all detected lines, for *clear* we reject the contributions of lines that result in a robot pose that is very different from the old pose. This is a form of particle filtering.

The reason is that in the state *clear* the robot is busy clearing the ball and is walking outside its goal area. The chance of making wrong detections of lines (e.g. mistaking the middle-line for a field-line) is larger relative to the chance of seeing correct lines.

8.4.3. Flags
Precise orientation and x and y values of the robot's pose are retrieved from the goal-lines. The only thing the flags need to do is making sure the overall orientation of the samples (resulting in choosing L_1, L_2, L_3 or L_4), is correct. The way this is evaluated is very similar to the method used for updating using the flags in the GT2003 software [7]. The angle to a detected flag is compared with the angle at which th e flag is expected to be seen. The way this angle-difference results in updating the robot's pose is different for *task-vision-request = guard* and = *clear*:

- *Position*. For every sample, if the difference in angles is larger than a certain threshold, the orientation of the pose is assumed to be wrong. A new orientation will be chosen randomly. Also the quality is adjusted.
- *Clear*. New orientations are not chosen randomly when the angle-difference is too large. Only the quality of the sample is adjusted.

The difference in *position* and *clear* in flag detection
In *position* a total new pose can be found very soon (a high update rate), while in *clear*, it is not possible to get a very different robot pose in a short time frame (a low update rate). The reason is (same as with lines), that in the state *clear*, the robot walks out of its own penalty area and the quality of the image processing is relatively bad.

8.4.4. Check reliability
The keeper self locator assumes that the robot is positioned inside its own penalty area. This assumption leads to serious problems if the robot actually is positioned somewhere else in the playing field. This is the reason we have to check whether this assumption still holds.

We use the following information about the goalie for checking the correctness of the robot-pose: when standing in its goal and doing a *search-lines* head-motion, the robot is expected to see three different lines and at least one flag during a certain time-interval (approximately 20 frames).

We use this information for checking the reliability in the following algorithm that keeps track of an array consisting of 5 parameters [L1, L2, L3, L4, F]. The values L_i and F represent respectively the 4 possible lines and the flags being detected. Their values can be between 50 and 200 and are changed in the following way:

For every frame
Decrease all 5 parameters by 1.
If a line or flag is detected at correct angle, the corresponding value is set to 200.

The reliability R is calculated at a product $R = L_1 * L_2 * L_3 * L_4 * F$.
Note that for the goalie, 4 of these 5 parameters tend to somewhere between 180 and 200 at all times, and R is high. For all other players, only 2 or 3 parameters are likely to be high simultaneously, and the reliability R will be significantly lower.

8.4.5. The parameters for the general self localization

When the goalie is away from its goal and has to go back to the own goal (*task-vision-request = return*) the general self locator is used. The algorithms of the general self locator is basically the same as the DT2004 self locator as described in chapter 4.2. Only a few things are slightly different from the DT2004 self locator if some parameters are set in *TaskSelfLocator.cpp*:

setparameters (useParameters = true, useGoalDistance = true, nrGoals = 4, goalPoints=false)

useParameters
useParameters only indicates whether or not all further parameters should be used. If *useParameters = FALSE*, the general self locator is identical to the DT2004 self locator.

useGoalDistance
If this parameter is true (as is the case in *task-vision-request = return*). The distance evaluation from a detected goal-percept is used to directly update the *x*- and *y*- position of the samples. In the DT2004 algorithm the measured goal-distance was not used, since at small distances from the goal, the error in the goal-distance evaluation is relatively large.

nrGoals
In the DT2004 algorithm, all update steps (odometry, line-points, goals, flags) where executed once. We have started to give a parameter that indicates how much influence the detection of one of these objects must have on the evaluation of the robot pose.

In the case of *nrGoals* = 4, the update process relying on goal-information is executed 4 times. Thus the quality of the samples is adjusted 4 times, every time a goal is detected.

goalPoints
The DT2004 self locator uses detected goalPoints (blue/green and yellow/green transitions) in the update process. In the modular image processing, detection of these points is not implemented. With goalPoints=false, the missing of these detected goal-points doesn't give problems.

9. Results

In this chapter we will test the performance of the goalie with the behavior based vision as described in chapters 7 and 8. We will compare its performance with the performance of the old goalie, which used a general vision system.

9.1. What do we test?

There are several techniques we have used in our system that contribute to a better performance. A schematic overview of all techniques and their contributions is depicted in figure 9.1. We have tested not only the performance of the entire new system compared with the performance of the old system. We have also tested for each independent technique how much it contributes to the overall performance.

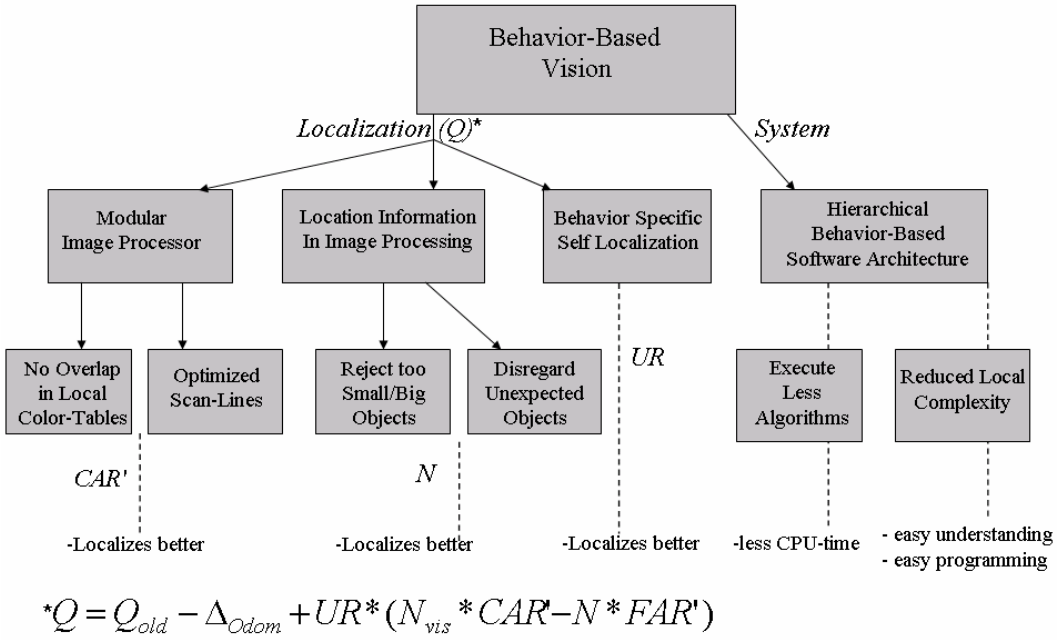


Figure 9.1. Schematic representation of the reasons why the performance of the new goalie should be higher then performance of the old one

The following items were tested:

Contributions individual parts

- An increase in performance due to modular image processing (Ch. 9.3).
- An increase in performance due to the use of location information (Ch. 9.4).
- An increase of available processing power (Ch. 9.5).

- Practical issues and limitations due to behavior dependence (Ch. 9.6)
- An increase in performance due to behavior specific self localization (Ch. 9.7).

Overall performance

- The performance of the new versus the old goalie (Ch. 9.7)

The performance increase of the image processing will be measured by evaluating a set of image-sequences (offline testing). The decrease in CPU-time will be measured by varying the

number of active algorithms and measuring the frame-rate. For the practical issues due to behavior dependence, we will describe some situations in which the new goalie has shown to have difficulties with self localization. The overall performance will be measured by comparing the performance of a goalie with a behavior based vision system with a goalie without behavior based vision system.

9.2. The testing environment

9.2.1. Lighting conditions:

We will do all tests under 5 different lighting conditions

- 1 Floodlight — 1 floodlight was placed above the playing field
- 4 Floodlights — 4 floodlights were placed above the playing field
- TL + floodlights — 4 floodlights and 2 TL-tube lights above the playing field
- Natural day light — all lights off and curtains open at daytime, no direct sunlight
- TL, floodl. + natural — all lights on and curtains open at daytime, no direct sunlight

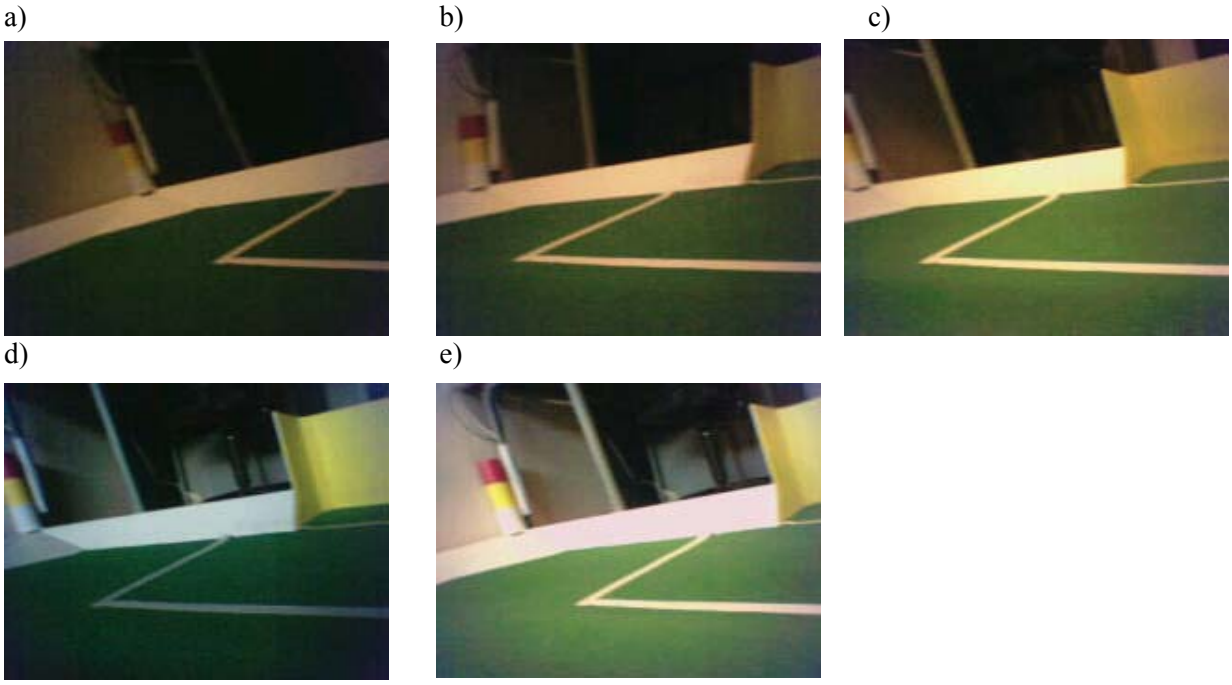


Figure 9.2. Lighting conditions; a) 1 Floodlight; b) 4 Floodlights; c) TL + floodlights; d) Natural day light; e) TL + floodlights+ natural day light

9.2.2. Image sequences for testing image processing algorithms

We will test the performance of individual parts of the system on the following sets of images.

- *Calibration set yellow-goal*: For 6 lighting conditions, we have set the robot at a distance of 2 meter from the yellow goal, facing the goal. We have taken an image set of 100 frames, with the head-control set to *search-for-landmarks*. In this setting the yellow goal and flags are supposed to be detected. The *Calibration set yellow-goal* is used for calibrating the color-tables.

- *Calibration set blue-goal*: The same as for the yellow goal, only now for the blue goal. In this setting the blue goal and flags are supposed to be detected. The *Calibration set blue-goal* is also used for calibrating the color-tables.

- *Test set goal-lines:* The robot stands in the center of the yellow goal executing the *search-lines* head-motion. In this setting the goal-lines and pink/yellow flags should be detected. Images were taken under 5 lighting conditions.

- *Test set various conditions*: Various lighting conditions, various behaviors. Not used for calibrating the color-tables

9.2.3. Robot setup and calibration of the color-tables

Behavior-based vision
For testing task-dependent vision, we use the robot with a behavior tree as described in chapter 9. The color-tables (lines.c64, bluegoal.c64, blueflag.c64, yellgoal.c64, yellflag.c64) are calibrated offline, on all calibration image sequences. Only one iteration step is used for calibration. The whole process of calibrating takes about 1 hour.

General vision
For testing general vision, we use the DT2004 behavior *player = goalie*. The DT2004 image processor and self locator are used.
We have calibrated the color-table, *Coltalbe.c64*, in the following way:

1. We have calibrated the color-table on all calibration image sequences in such a way that at least one true positive was detected in all images. All the green is first calibrated as green. All the green/blue overlap is later relabeled blue.
2. Then, colors were removed from all sequences that were resulting in false percepts
3. In a third step, the color orange was calibrated on an image sequence of pictures of the ball, at various distances and varying lighting conditions.
4. In a fourth step, again the color-tables were tested and optimized on the whole calibration set; colors were removed and/or added.

The whole process was repeated 2 times, adding or removing only a little color here and there in the color lookup table. The whole process takes about 2/3 hours.

9.3. Impact of modular image processing

We have compared the performance of the old image processor (DT2004) with the performance of a "general" version of the new image processing (Task Image Processor): Now all available algorithms are executed in every frame (*ownGoal*, *opponentGoal*, *ownFlag*, *opponentFlag*, *Lines*). The two image processors are different due the use of object-specific 3-color color-tables in stead of a single general 10-color color-table, and due to the use of object-specific definition of scan-lines instead of a single general grid used for all algorithms.

9.3.1. Measuring the performance

The performance is measured in the following way:
The performance of flag- and goal detection is measured on both the calibration sets and the "various positions" set from the log-files. For all sequences we have counted the number of correct flag -and goal detections (true positives) and the number of incorrect flag- and goal detections (false positives).

For every sequence we have evaluated the correct acceptation rate CAR (N true positives / total percepts visible) and the false acceptation rate FAR (N false positives / total percepts visible). The Total Percepts Visible is estimated for each log-file. The TPV depends on the number of frames in a log-file for a certain position of the robot as well as the head-motion carried out.

9.3.2. Measurements

In tables 9.1 to 9.3, a summary of the results of tables A.1-A.4 (Appendix A) is presented.

Table 9.1.Performance on detecting flags. The FAR is given as percentage as well as the total number of false detections (in brackets)

	CAR		**FAR**	
Name Set	DT2004	**Task**	DT2004	Task
Calibration Set "Yellow Goal"	45 %	**77 %**	2.5% (9)	0.3% (1)
Calibration Set "Blue Goal"	35 %	**73%**	0.28% (1)	0
Test set "Various Positions"	43 %	**73%**	9.3% (28)	5.3% (16)

Table 9.2. Results on detecting goals. The performance of the goal detection does not increase significantly with the modular image processor

	CAR		**FAR**	
Name Set	DT2004	Task	DT2004	Task
Calibration Set "Yellow Goal"	41 %	44 %	2.78 (10)	0.3 (1)
Calibration Set "Blue Goal"	38 %	42%	0.56 (2)	0
Test set "Various Positions"	30 %	60%	1 % (3)	3.7 %(11)*

** 10 of the 11 errors are due to a blue bag in the playing field identified as a goal.*

Table 9.3. Results on detecting Lines

Name Set	DT2004	Task
Calibration Set "Lines"	40 %	**95 %**

9.3.2. Summary
We see that the performance of the task image processor is dramatically higher than the performance of the DT2004 (general) image processor.

- The CAR of the flag detection dramatically increases from 40% to approx. 75 %
- The CAR of the line detection even increases from 40 to 95 %
- The FAR of the flag detection decreases, only errors in objects outside the field remain
- There is still room for improvement for the goal detection algorithms

With these results, the factor CAR in formula (5.8) is not far from the theoretical maximum of 100 %. Without using any white-balancing or auto-calibration, we obtain a very reliable and robust image processor.

9.4. Using location information

9.4.1 Using distance information to disregard far/near objects
The OwnFlag (yellow flag) algorithm was tested on two acceptations sets:

- One set of the goalie guarding its goal
- One set of players standing in the field facing the yellow goal

The algorithms were tried twice: once while accepting flags of all sizes and the second time while accepting flags with a minimum size only. The results can be found in tables A.5 and A.6. (Appendix A). A summary is given in table 9.4.

Table 9.4. Using distance information

	Accept all flags		**Reject small flags**	
Image-set	True positives	False positives	True positives	False positives
Goalie in goal	67	6	57	0
Player in field	85	1	35	0

From table 9.4. we can determine that when the goalie is using distance information (this holds for any other player which' own position is defined very well), this can lead to a significant decrease in the number of false positives without dramatically affecting the number of true positives.
We also can see the danger of using distance information: it can dramatically decrease the quality of image processing if it is used too reject small flags when small flags are expected to be seen.
Consequently, distance information should not be used unless the need for a small FAR is very high and the own pose is known very accurately.

9.4.2. Disregarding unexpected objects

We will run some algorithms in the acceptation set *stand in goal* (under all lighting conditions). We will run the *OwnFlag(true), OwnGoal()*, *OpponentGoal()* and *OpponentFlags(false)* algorithms and count the number of true and false positives.
The results can be found in table A.7 (Appendix A); a summary is presented in table 9.5.

Table 9.5. Influence of disregarding unexpected objects

	Accept all objects		**OwnFlag only**.	
Test Set	True Positives	False Positives	True Positives	False Positives
Goalie in Goal	84	10	57	0

We can see that searching only for the objects that are expected to be seen (correctly) can greatly reduce the number of false positives (and thus the FAR), while still a reasonable number of true positives is detected.
Consequently, disregarding information (using information only of objects that are likely to be seen correctly) can increase the performance. Prerequisite is that the own pose is known accurately.

9.5. Processing power

We have executed several image processing algorithms for a varying number of times per time-step and have evaluated how this influenced the total frame-rate of the system. Results can be found in tables A.8 and A.9. (Appendix A).

From these results we can conclude the following:

- The normal frame rate of the system is 30 frames/s
- Modular image processing (when executing all algorithms) requires approximately the same processing power as the old general image processing. There is not much overhead.
- If we would use twice the current complexity in image processing, the frame-rate would go down to approximately 25 frames/s. With this, playing soccer is still possible.

- However, if we use specific vision (only detecting e.g. line-points and flags), we could use algorithms with a complexity of approximately 5-10 times the current complexity.
- Using the Hough transform is a relative costly operation; using it for detecting lines using 30-40 line-points (in one image) is now possible. Applying the Hough transform on 200- or more points, still slows down the system.

9.6. Practical issues and limitations due to using behavior specific algorithms

There are some practical issues that arise when using a behavior based vision system in which information is disregarded and local loops are made possible. These issues do not arise in a system with a general vision system. We will discuss two of those issues.

9.6.1. Checking the viability of local loops is not a real-time process

A local position loop may use the information that the robots position is roughly correct. For this we need a checking mechanism that robustly detects when the own position is wrong. To detect when and only when the own position is wrong, the evaluation of many subsequent frames (and hence many seconds) is required. If too few frames are evaluated, the chances are high that the robot will wrongly assume he has lost its position. This delay of seconds negatively influences the performance of the robot when local loops are used more than rarely. We can see examples of this when the goalie does not manage to return to its goal in one try. If the robot thinks it has returned but in fact it has not, the robot starts guarding the wrong position and isn't making another attempt to return to its goal for at least another 5 seconds. If the return-to-goal behavior with the return-to-goal local loop vision system makes many mistakes, the process of returning to the own goal can take a significant time.

9.6.2. Disregarding information when information is scarce

We have implemented a self localization method for the goalie using only the 4 field-lines of the goal and the 2 own flags, making the assumption that the goalie can always see the field-lines and sees one of the flags once in a while. With this vision system, the goalie can very robustly localize itself when standing in the centre of the goal (see the test *localize in penalty area* in chapter 9.7).

The goalie, however, is not always located in the centre of the goal; it also does actions in which it is located elsewhere, where the field-lines might not always be visible. To overcome those situations, we have built specific methods for self localization that use particle filtering at a low update rate, in order to reduce the chance that the robot gets disorientated while doing an action. We have experienced that these methods are well capable of bridging short times in which the robot can't see much. When the vision input is neglectible over too long a time, however, the quality of self localization can be seriously reduced. For example:

- Good localization: clearing a ball.
When the ball comes inside the penalty area, the robot goes to the ball, kicks it away and returns to the center of the goal (fig 9.3a). Only for a short time (5-10 seconds) the robot relies more on odometry than on the field lines. The robot bridges this period very well.

- Bad localization: ball near the field border.
There is, however, a situation in which problems arise. This situation can be seen in figure 9.3b. If the ball is near the field border, the robot positions itself between the ball and the goal, resulting in the robot standing outside the centre of goal, facing the side line. In this pose the robot can't see many proper lines, necessary for his local localization loop. We have

experienced that the goalie was not accurately blocking the opening to the goal when the ball was near the field border for a longer period of time.

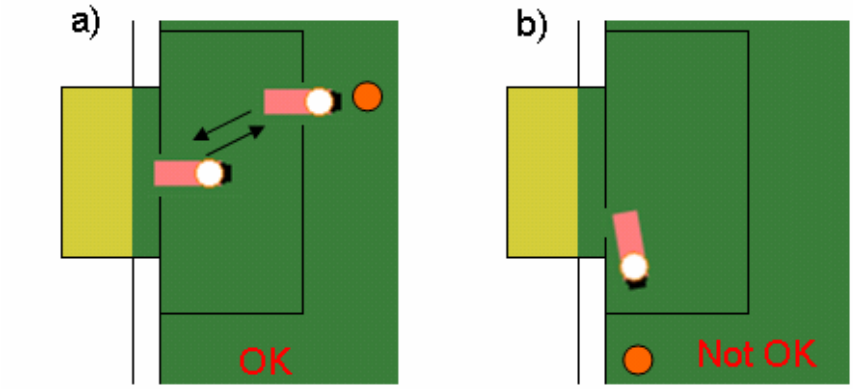


Figure 9.3. Two situations with limited input for the goalie's vision system; a) the goalie clears a ball and then returns. The goalie is only away from the center of the goal for a short while, there is no problem with localization; b) the ball is located near the border, the goalie positions itself between the goal and the ball. For a long time, the robot can not see enough proper lines and problems with self localization arise.

For a goalie to robustly guard its goal also when the ball is near the field border, more information should be taken into account for self localization. E.g. the detected field border-lines could be used on top of the detected field-lines and flags. In this way, the goalie would have enough information, wherever he is located.

9.7. Tests in the real world

We have tested the performance of the new behavior-based system. For being able to distinguish between the performance increase due the modular image processor (which is behavior-independent) and the performance increase due to behavior-specific algorithms, we have performed tests with three vision system variants.

9.7.1. Vision systems used

The following vision systems were used:

DT2004.	The (old) general image processor with a general self locator
Striker	All algorithms of the modular image processing with a general self locator. This solution is used by the defender, striker1 and striker2
Goalie	Modular image processing with behavior-dependence in image processing as well as self localization for the goalie. When guarding the goal, only the own flag and the lines are used. When returning to the goal all objects are used for localization; the weight of a detected own-goal is higher than the weight of other objects

9.7.2. Scenarios

Localize in penalty area

The robot is manually put into the penalty area and has to return to the return spot as quickly as possible. The purpose of this test is mainly to evaluate the difference between behavior-specific (goalie) and general self localization (striker).

Return to goal

The robot is manually put onto a predefined spot in the field (P1 or P2) and has to return to the return spot as quickly as possible. This test has two purposes: first, it is used to evaluate the effect of behavior switching on the overall performance. Second, it can show the advantages of using modular image processing over general image processing.

Clear ball
The robot starts in the return spot, has to clear the ball from the penalty area and return to the return spot as quickly as possible. The ball is placed back when the return spot has been reached. The purpose of this test is to evaluate the robustness of the vision systems in real action.

Clear ball with obstacles on the field
We have performed the clear ball test also when many objects and some robots are placed on the field. The purpose of this test is to show the impact of using location information when the environment becomes more natural (complex).

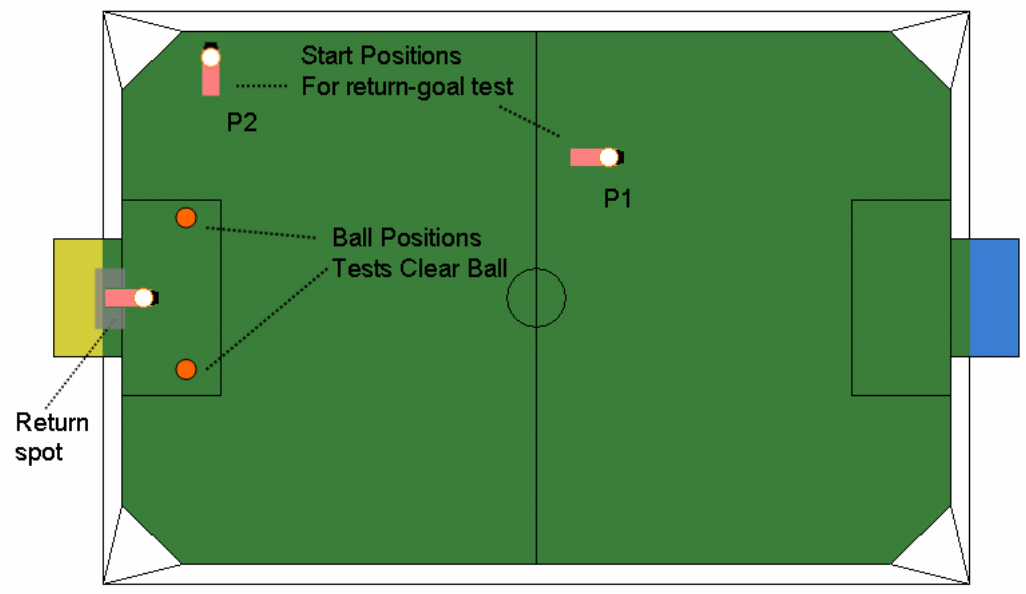


Figure 9.4. The setup for all tests in the real world.

9.7.3. Results

Localize in penalty area
The results of the test *localize in penalty area* can be found in table A.10 (Appendix A). The results are also represented in figure 9.5. One can see there that the DT2004 image processor manages to localize the robot correctly only in a limited range of lighting conditions (not with TL light only). Both the vision systems that use the modular image processing (striker and goalie), manage to localize the robot correctly under a wide variety of lighting conditions. The behavior specific self localization of the goalie (which localizes on the field-lines mainly), allows for an additional 50 % increase in performance over the vision system of the striker. Its localization on field-lines is more robust and allows for quicker relocation. We have also tested the general vision system when only yellow flags and lines were used as its input. This resulted in a lower performance.

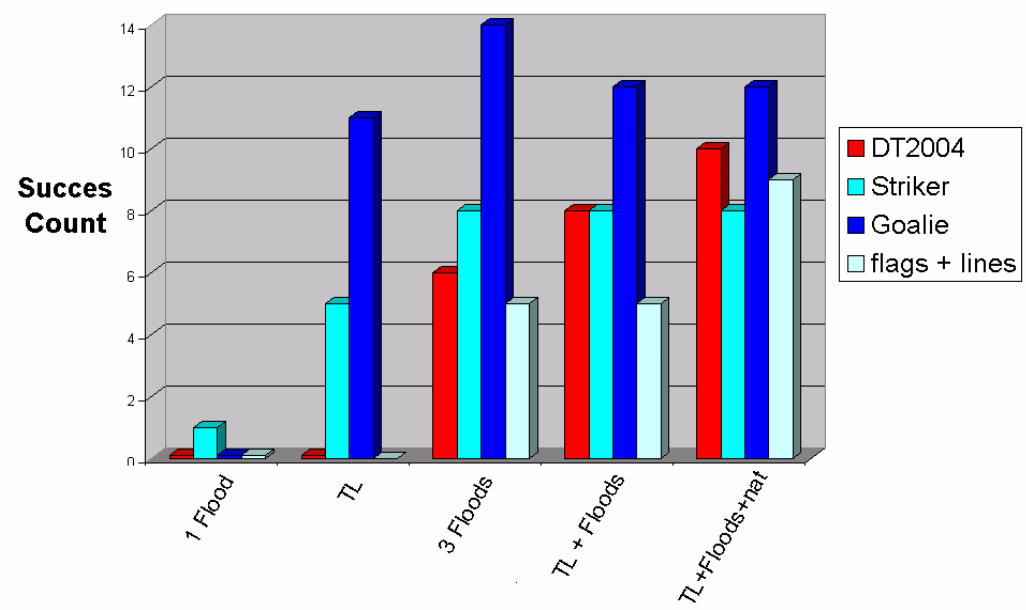


Figure 9.5. Results for localization in the penalty area. The number of times the robot can re-localize in the penalty area within 2 minutes. The DT2004 vision system cannot localize when there is little light (TL). Using behavior specific self localization increases the performance of the modular image processor with an additional 50 %.

Return to goal

The results of the return to goal test can be found in table A.11 (Appendix A). The results are also shown in figure 9.6. Both the striker and the goalie (using modular image processing with multiple color-tables) can find their way back to the goal much better than the DT2004 robot with its general vision system. Furthermore, it is interesting to see that there is no significant difference in performance between the striker and the goalie. The negative impact of switching vision-systems appears to be limited.

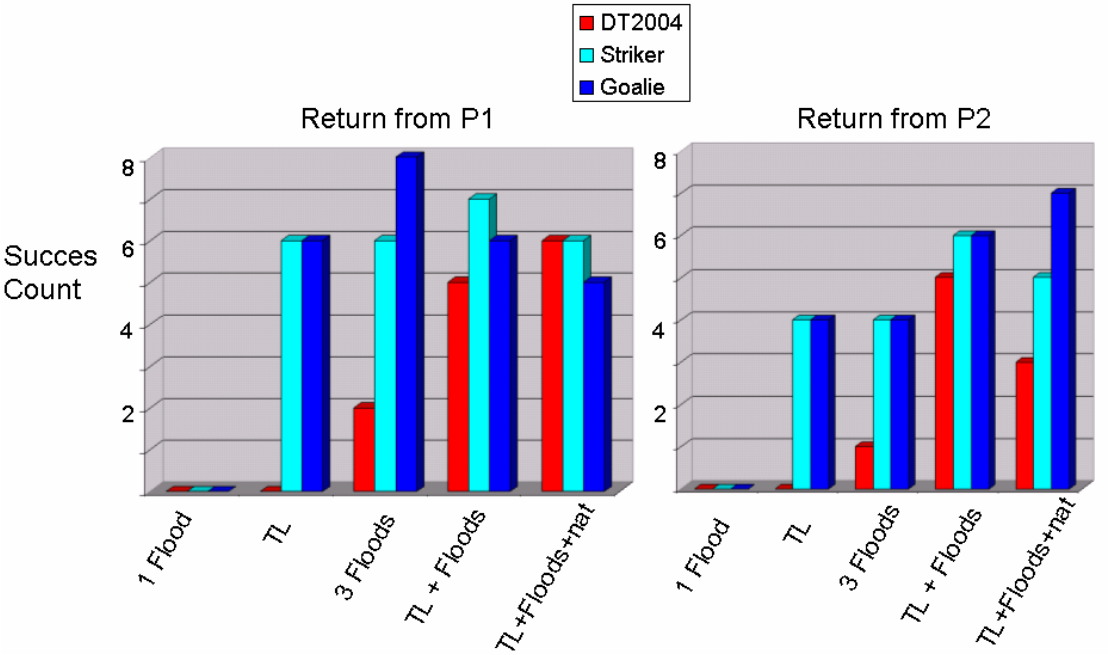


Figure 9.6. Results of the return to goal test. The robot has to return to its own goal as many times as possible within 3 minutes. The modular vision systems work significantly better than the general vision system. There is no significant difference in overall performance between the striker (no behavior-dependence) and the goalie (behavior dependence).

Returning from position p2 is more difficult, both for the striker and the goalie, however, for quite different reasons.
For the striker it is difficult because it takes quite some time before it realizes it is located wrongly.
For the goalie it is difficult because the robot doesn't always manage to return to goal exactly in one try. The robot might think he is inside the penalty area, while he can be still outside it. He then has to do the process of detecting that its position is wrong and relocate him self once more. This process costs time as we have already explained in chapter 9.6.1.

Clear ball
The results of the *clear ball* tests can be found in table A.12 (Appendix A) and are presented in figure 9.7. The vision system with behavior-dependence (goalie) works approximately 50% better than the system without behavior-dependence (striker), as we have also seen in the *localize in penalty area* test. A very interesting thing is, that for the 1 floodlight lighting condition, the goalie was able to clear the ball 5 times before loosing its position. In this condition there was not enough light to detect a flag. The goalie could clear the ball 5 times localizing purely on the field-lines (after 5 times, his orientation had shifted 90 degrees).

The performance of the goalie vision system is not yet totally independent of the lighting condition. The reason is that the yellow flag detection makes use of a color table containing yellow/white and pink and there is an overlap between yellow and white; this overlap becomes most apparent when there is not much light (TL).

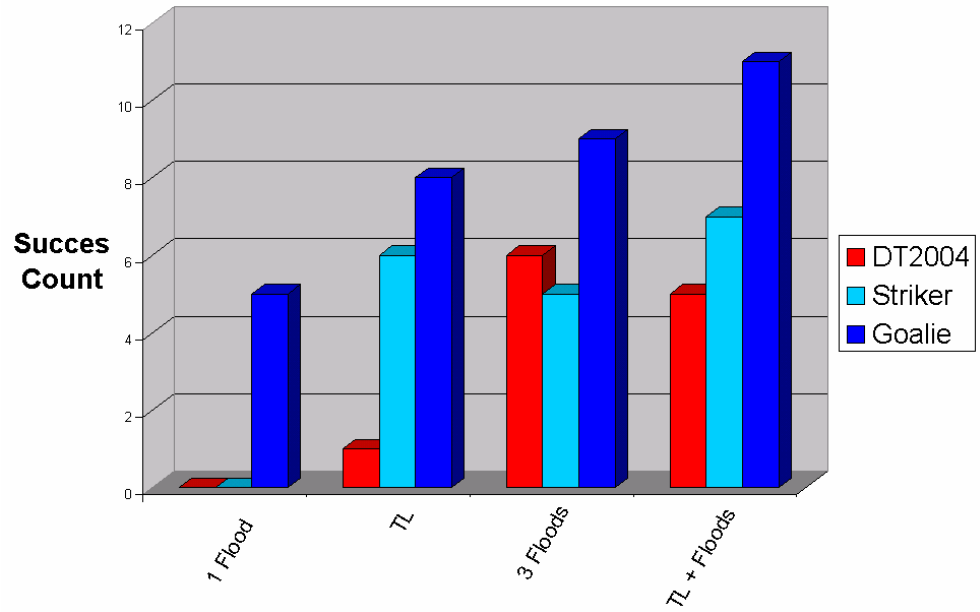


Figure 9.7. Results of the clear ball test. Both the striker and the goalie vision systems (using modular image processing) are more robust in a larger variety of lighting conditions than the DT2004 vision system (using a general image processor with only 1 color table).
The goalie's self locator, using detected lines and the yellow flags, works approximately 50 % better than the striker self locator which locates on line-points and all flags and goals.

Clear ball with obstacles on the field

The results of the clear ball with obstacles test can be found in table A.13 (Appendix A) and in figure 9.10. Here a very big advantage of behavior-dependence within the vision system appears. The striker vision system localizes on 4 flags, 2 goals and lines. When many obstacles in the field are placed, (figure 9.8) the chance of making errors in the image processor increases. Goals and flags might be detected in players and other objects. At the same time, the chance of detecting the blue goal and flags decreases, due to objects blocking the sight. For the striker vision system, it becomes nearly impossible to localize when many obstacles are located in the field.

The performance of the goalie vision system, which uses only field-lines and the yellow flags for localization, is far less influenced by the fact that strange objects are placed in the field. One can also note that in this test the DT2004 vision system actually performs better than the striker vision system. The reason is that the DT2004 system (without modular image processing) detects fewer objects, and thus also detects less false objects than the striker vision system.

Figure 9.8. Camera image of the robot when many obstacles are placed in the field.

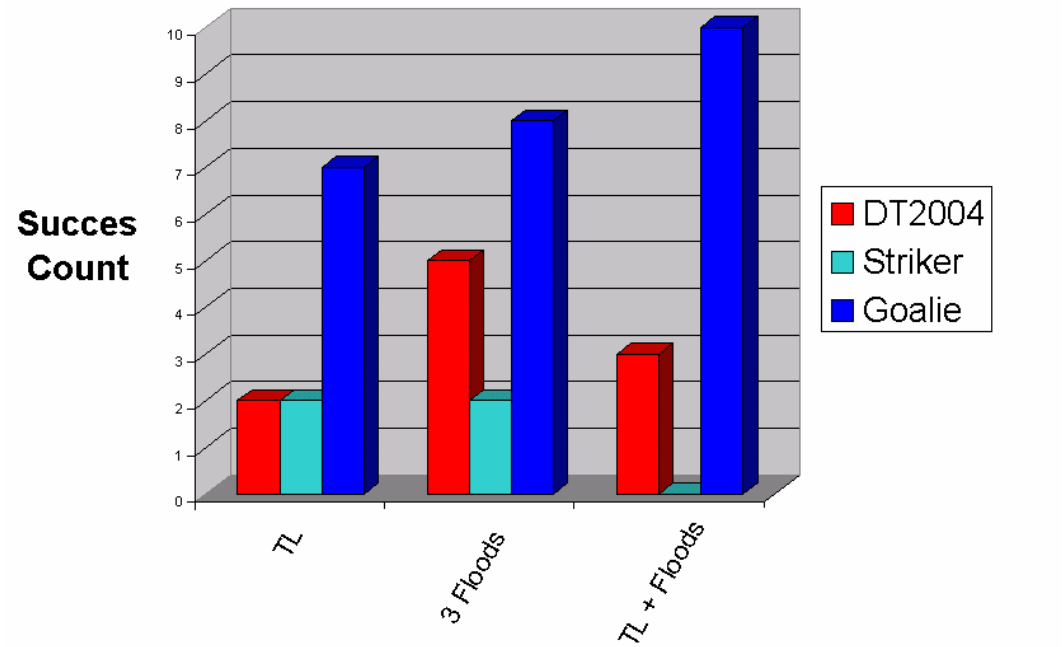


Figure 9.9. Results of the clear ball with obstacles on the field test. The goalie vision, which uses location information to disregard blue flags/goals and only detects large yellow flags, is very robust when many unexpected obstacles are visible in or around the playing field.

10. Conclusions

We have implemented a behavior-based vision system for a goalie.
To enable behaviour dependence, we have implemented a modular image processor which uses 5 object-specific 3-color color lookup tables in stead of a single general 10-color color lookup table and we have built in some other features than can be used to realise behavior dependence. They are:

- behavior-coupled location information for rejecting unexpected objects
- behavior-coupled location information to reject small objects
- behavior specific self localization

We have tested this system and came to the following conclusions:

- Using the modular image processor, the correct acceptation rate of the image processing can rise from approximately 40 % to approximately 75 %, under a wide variety of lighting conditions, and the false acceptation rate is reduced. Moreover, it takes significantly less time to calibrate the five 3-color color lookup tables than the single 10-color lookup table

- For a goalie guarding its goal under normal lighting conditions, using in addition behavior-specific self localization, leads to an extra 50 % performance increase

- Under special conditions, such as when many obstacles are visible in the field, the use of the behavior-dependent location information for rejecting unlikely objects often is the essential factor between being able to localize and not being able to localize.

- The greatest disadvantage of using behavior-dependence in a system is that one has to design when the vision-system changes its state. For this one has to use a mechanism for detecting local loops. The detection that one is in a wrong local loop and consequently switching back and forth between behaviors, consumes time and results in a decrease of performance. We have proven that the negative impact of the use of behavior-specific self localization can be kept relatively small for a goalie: the positive impact of using location information and behavior-specific self localization well outweighs the negative impact of every now and then being in the wrong local loop.

- Due to the reduced complexity of the software, the impact of a hierarchical behavior-based design is very large. It has proven that it is possible to make huge steps in improving the overall system within a few months. In fact, the system is changed from one single big sense think act loop to a hierarchy of small sense, think, act loops. This, however, makes it necessary that the software engineers must have knowledge of the entire content of these “sense, think, act” loops and hence should have (some) skills in image processing, AI, and motion control.

- Using modular image processing does not result in a significant overhead. Using it to implement a general vision system, then algorithms consuming about 2 times the currently used processing power are possible. However, when using only 2 or 3 specific algorithms as is done in behavior based vision, then it is possible to use algorithms that consume 5-10 times the CPU-time of current algorithms.

11. Discussion and recommendations

11.1. Behavior based vision for the defender/striker

We have shown and proven that a goalie with a behavior-based vision system can work significantly better than a goalie with a general purpose vision system. A goalie, however, is quite different from other players. We will summarize the differences and similarities between the goalie and the other players in tables 11.1 and 11.2.

Table 11.1. Differences between goalie and other robots

Goalie	**Other robots**
Well defined own location (penalty area)	Less defined location (own or opponent half)
Well defined heading (facing the field)	No well defined heading
An accurate robot-pose (x,y,θ) evaluation is essential for performance. Otherwise the robot does not block its goal	Absolute robot-pose is not essential. A good heading (θ) evaluation is enough for kicking in the right direction

Table 11.2. Similarities between goalie and other robots

Goalie	**Other robots**
Well defined actions: guarding goal, clearing ball, or returning to goal	Well defined actions: search ball, handle ball, or position
When clearing the ball, the performance of the image processing tends to be low	When handling the ball, the performance of the image processing tends to be low
The goalie can accurately detect its own flags and lines. When returning the goalie likely detects its own goal	The striker can accurately detect the opponent goal and flags. The defender can accurately detect the flags standing in the centre of field
CPU-time is limited	Idem
Simplicity of behavior is crucial	Idem

Specific self location not applicable
A specific analytic self locator can make a fast and accurate robot-pose evaluation from only a few percepts. For the field robots however, it is not known exactly what objects are likely to be seen, since the robots have no well defined default location. Moreover, a real accurate robot-pose evaluation is not required for field players. For them it makes much more sense to use a probabilistic (Monte-Carlo) method for self localization, providing robustness against false positives. All behaviors other than the goalie standing in goal should make use of the general self locator with parameters.

Note, however, that the possibility of specific self localization for the field players should not be excluded in the process of development. If someone is researching localization methods (see chapter 11.2) fundamentally different from the one now currently used, it makes sense to implement these in an entire new self locator in stead of making the general self locator more and more complex.

Other implications are applicable.
All other implications of the behavior-based vision system are very good applicable to the entire robot system:

- *Modular image processing* with multiple color-tables is applicable to the whole system. We have implemented this and the robustness of the localization of the striker and defender has dramatically improved.
- *Using variable weights for objects in the general self locator* is applicable to the entire system. As we have given more weight to a detected own goal for the returning goalie, we can give more weight to a detected opponent goal and flags for a striker.
- *Variable update rate particle filtering*. For the goalie we have used particle filtering with a low update rate for the goalie when clearing the ball. This could also be done for the field-players. When the player walks around or searches the ball, it is likely to see many objects and the quality of image processing is high. When the robot is only looking at a ball or is handling the ball with its head, the robot is unlikely to detect many objects and abundant use of image processing can only lead to a decreased quality; the general self locator should be augmented with a parameter that can set the update rate and can trigger the use of particle filtering, dependent on the activity..
- *A behavior based hierarchical architecture* leads to an understandable system. The possibility to easily switch on/of algorithms for image processing makes debugging and developing image processing algorithms much more easier.
- The possibility of using only the most critical image processing algorithms when *processing power* is limited can be essential for the entire system.

Our main recommendation for developing a behavior-based vision system for the other players is to further develop the general self locator adding some parameters. With a further developed localization on lines, it might even be possible to use the general self locator also for the goalie. Developing the general self locator could be a bachelor assignment.

11.2 Modular image processing and color-independent algorithms

Shape-based algorithms are the next step.
The modular image processor as described in this thesis makes use of specific gridlines to reduce the number of pixels to be processed and five 3-color color-tables in stead of a single 10-color color-table. With this, the weight of the classification process has already largely moved from color-segmentation to shape-evaluation (with all advantages, such as robustness in various lighting conditions). The next logical step is to entirely go to shape-based algorithms. Instead of retrieving candidates by doing segmentation on absolute colors, candidate objects can be found by segmenting on relative colors (edges). An image processor algorithm based on shapes does not need color-tables. And calibration of color-tables will not be necessary.

Implementing shape-based algorithms in a modular image processor
In the DT2004 software, developing an entirely new (better) method for image processing was difficult. A new image processor had to replace the entire old system, and thus had to compete in performance in all tasks of the image processor (detecting flags, lines, goals, ball, etc).

A modular image processor serves as an excellent platform for researching and implementing shape-based (or other) algorithms. The reason is that one can easily replace a grid-based algorithm (e.g. ball detection) by a shape-based algorithm, leaving all other algorithms intact (see figure 11.1). One does not need to build an entire new vision system before a new technique can be implemented and tested.

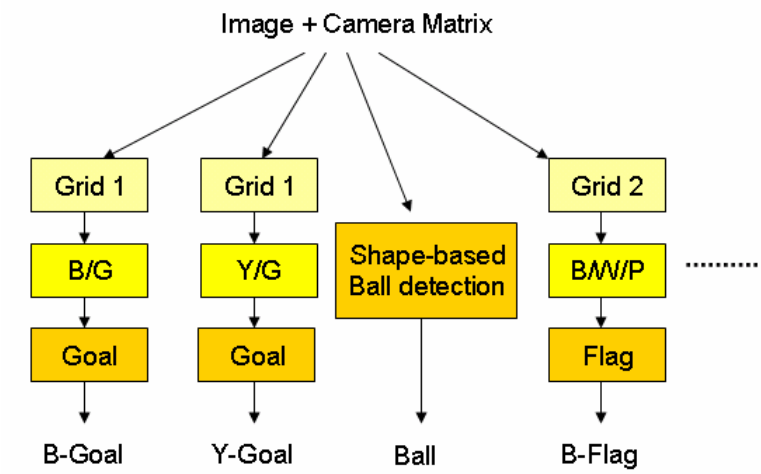


Figure 11.1. Modular image processing allows the total system to remain working when old algorithms are replaced by new ones.

Building shape-based image processing algorithms and implementing them into the system are typically bachelor assignments.

11.3. Other fields of research and implementation in a behavior-based architecture

11.3.1 Using SIFT features for localization.

The vision-based self localization in the current system is based on a lot of expert knowledge. The programmer has decided that it is good to localize on flags, goals and lines. Also the programmer has decided how to best detect these objects. The algorithms use color-tables that have to be calibrated by an expert (human). Using this expert-based vision system leads to problems if the surroundings of the soccer game are different than the designer (expert) has designed for. If objects in the audience look similar to flags or goals they can be mistaken for the expected objects. Playing in a surrounding without the expected objects (e.g. without flags) is not possible without writing entire new algorithms for image processing.

A very interesting field of research is using self localization relying less on expert knowledge and relying more on characteristics of the playing field the robot determines himself. A system in which the robot doesn't necessarily localize on goals/ flags, but in which it localizes on whatever features specific to the environment (can be any object in the room). An example is the method of using SIFT features [8]. The robot calculates a set of distinctive invariant features from images for determining its position and orientation. When a new type of field is used, the robot just has to calibrate what features belong to which robot-pose, and the robot can localize.

A huge issue in a method as SIFT is, that it requires much more processing time to calculate the SIFT features in an image than it takes for detecting known flags or goals with the current algorithms. A system using SIFT features for localization could not operate in real-time (30 frames/ s) with the processing power available in current robots. Replacing the old general vision system with a SIFT-based vision system therefore is no option yet.

However, there is no reason why the robot should have to evaluate these SIFT features 30 frames per second. SIFT features could possibly be implemented in the current system if we

would identify two different problems in the localization process: one for finding the general robot pose, not in real time, and one for only updating it in real time.
The SIFT-based vision system would only be used to generate the overall pose once in a while and is allowed to consume much processing time. A classical method for tracking the robot-pose, using e.g. only odometry and line information could be used for actualising the robot pose in real-time. A schematic representation of the two separate systems and their characteristics can be found in table 11.3.

Table 11.3. Dividing the localization process in 2 steps. One step for finding the general pose (x,y,θ) if the pose was not known. This process only has to be executed once in a while. It doesn't matter this process costs much CPU-time. The second step is only for updating the pose between two evaluations of the general pose

Self Localization A	**Self Localization B**
Find general pose	Track pose
Old pose assumed incorrect	Old pose assumed correct
New pose anywhere in the field	New pose near the previous pose
Executed once in 5 or 10 seconds	Executed 30 times /s
Using e.g. SIFT features, using much CPU-time	Odometry and lines for updating pose, requiring little CPU-time

The behaviour-based vision system as described in this thesis could be an excellent platform for implementing two different methods of self localization in a real-time robot system. Implementing SIFT features for AIBO localization could be a very interesting masters' thesis.

11.3.2. Cooperation between robots

One of the things the robots don't do at this time is cooperating. The AIBO's currently don't pass, they only run to the ball and kick it in the direction of the opponent goal. The soccer game would be much more interesting if robots would pass to each other and really start working as a team. There are two major things that need to be developed before real passing is possible for robots operating in the real world (in the soccer simulation league, robots started passing years ago). The vision system needs to be improved for accurately detecting fellow players. When this is done, the robot will need to have decision mechanisms for deciding when to shoot and when to pass.

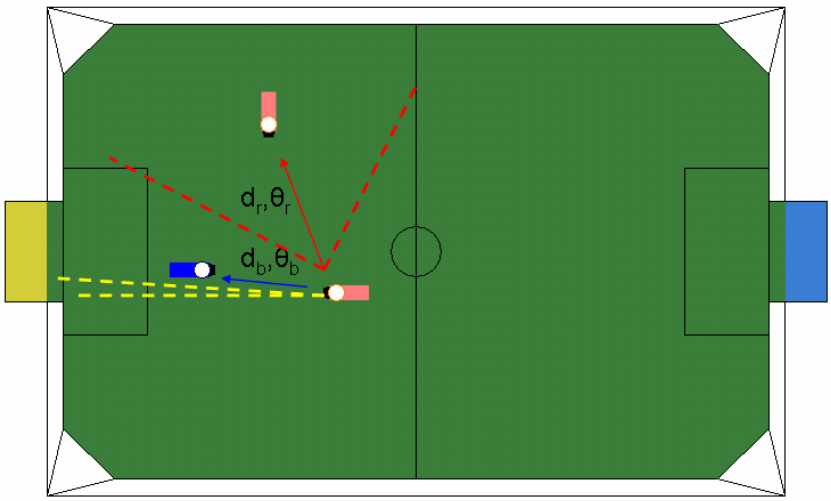


Figure 11.2. Parameters relevant for passing. The robot needs awareness of the position of its fellow player (d_r,θ_r) if it wants to pass to him rather than to shoot at goal

1. Vision: location of players.
The current system only evaluates the own robot's pose and detects obstacles (no green field indicates an obstacle). Therefore the only thing the robot can do is walking to a position or a ball with or without obstacle avoidance and kicking the ball to the opponent goal. The robots in the DT2004 software are very poorly capable of detecting opponents and fellow players, which is prerequisite for passing. The first step toward cooperation would be developing image processing algorithms for accurately detecting players. Developing these algorithms should not form any problems, since the robots wear color-marked shirts (dark blue or red). In the future it might be good to know the distances and angles to both the players and opponents (d and θ in figure 11.2). In the first step it might be good enough to just evaluate free angles to the goal (yellow lines in figure 11.2) and the free angle to a fellow player (red lines in figure 11.2).
In the Dutch Aibo Team software, basic algorithms for detecting players are already available and thus all the interfaces are already defined (figure 3.3).

2. Behavior: deciding when and where to pass.
When the robot is capable of detecting opponents/ fellow players, the robots must decide when to shoot to goal and when to pass to a fellow player. First, one can use very simple decision trees such as: if the free angle to an opponent goal is small and the free angle to a fellow player is large, pass to the player, shoot at goal otherwise. In the next step one can use more advanced decision trees, taking into account the position of the player, the distance and angles to the opponents and fellow players, and the free angle to the goal. In this case the possible decision space will become very large and one might consider using learning algorithms (such as reinforcement learning) for deciding when and where to pass dependent on the input parameters. In the case of using reinforcement learning, a good simulator of the robots would be a prerequisite. Learning when to pass might be a very similar problem to learning how to dribble [10].

References

[1] http://www.robocup.org

[2] http//aibo.cs.uu.nl

[3] http://www.ais.fraunhofer.de/GO/2004/

[4] Thomas Röfer and Matthias Jüngel (2004) “Fast and Robust Edge-Based Localization in the Sony Four-Legged Robot League”. In: 7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences). Lecture Notes in Artificial Intelligence, Padova, Italy. Springer, to appear.

[5] Bram van Driel. A Self-Calibrating Vision System for Soccer Playing Robots. MSc thesis, Quantitative Imaging Group, Faculty of Applied Sciences, Delft University of Technology, Delft, The Netherlands, August, 2004.

[6] Maayan Roth, Douglas Vail, and Manuela Veloso. A world model for multi-robot teams with communication. In Proceedings of IROS'03, submitted for publication.

[7] Thomas Rofer, Oskar von Stryk, Ronnie Brunn, Martin Kallnik and many others. German Team 2003. Technical report (178 pages, only available online: http://www.germanteam.org/GT2003.pdf)

[8] David G. Lowe. "Distinctive image features from scale-invariant keypoints" International Journal of Computer Vision, 60, 2 (2004), pp. 91-110

[9] R.Pfeifer and C.Scheier. Understanding Intelligence. The MIT Press, Cambridge, Massechussets, 1999, ISBN 0-262-16181-8

[10] Martijn Geers “Teaching Soccer Skills to a robot” MSc Thesis, Quantitative Imaging Group, Faculty of Applied Sciences, Delft University of Technology, Delft, The Netherlands, December, 2003.

[11] S.Thrun. Particle filters in robotics. In the 17th Annual Conference on Uncertainty in AI (UAI), 2002

[12] S.Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust monte carlo localization for mobile robots, Journal of Artificial Intelligence, Vol. 128, nr 1-2, page 99-141, 2001. ISSN:0004-3702

[13] T.Rofer and M. Jungel. Vision-based fast and reactive monte-carlo localization. In The IEEE International Conference on Robotics and Automation, pages 856-661, Taipei, Taiwan, 2003.

[14] Robot soccer team Clockwork Orange. http://www.ph.tn.tudelft.nl/~phrobosoc

Appendix A. Tables and measurements

A.1. Measuring performance modular image processor.

The quality of the old image processor (DT200) with one color table is compared with the performance of the task image processor (Task) using 5 independent color tables.
We have calibrated all color tables off-line using both the Yellow Goal and Blue Goal calibration set.

Table A.1: Calibration set yellow goal

Each logfile contains +- 50 goals, 60 goals in a total of ~100 frames

	DT2004					Task				
	Flag		Goal			Flag		Goal		
Log File	Correct	False	Correct	False	Reason	Correct	False	Correct	False	Reason
1Flood	20		13			33		18		
4Floods	45		12			60		32		
Natural	21		33	10	blue Goal	28		15		
TL	25	8	25		Yellow Flag	41	1	37	1	Yellow Flag+Goal
TL+floods	29	1	26		Yellow Flag	66		17		
TL+floods+nat	23		14			51		12		
Total	163	9	123	10		279	1	131	1	
AR	45.28	2.5	41	2.78		77.5	0.278	43.67	0.278	

Table A.2: Calibration set blue goal

Frames containing +- 50 blue goals, 60 blue flags, 100 frames per sequence.

	DT2004					Task				
	Flag		Goal			Flag		Goal		
Log File	Correct	False	Correct	False	Reason	Correct	False	Correct	False	Reason
1Flood	3		1	1	yellow Goal	32		5		
4 Floods	41		22			59		31		
Natural	26		8	1	Blue Goal	40		7		
TL	29	1	8			58		32		
TL+Floods	12		46			44		18		
TL+flood+natural	16		29			31		34		
Total	127	1	114	2		264	0	127	0	
AR (in %)	35.28	0.278	38	0.56		73.33	0	42.33	0	

Table A.3. Test set various positions

log files of various sizes, various head-motions and at various lighting conditions
In total approx 300 frames with goal/flag

	DT2004					Task				
	Flag		Goal			Flag		Goal		
Log File	Correct	False	Correct	False	Reason	Correct	False	Correct	False	Reason
1Light	40	1	26	0		71	4	42		Pink chair outside field seen as blue flag
4LightsKeeper	8	0	0	0		11		3	11	Blue Bag in Playing Field seen as blue goal

010Fast	54	8	2	0		54	1	39		
Face yellow goal no head-motion	10	9	20	0	Yell Flag	24	5	44	0	Blue Flag
210Fast	17	8	42	0		29	6	26		
212Fast	2	2	0	3		29	0	24	0	
Total	131	28	90	3		218	16	178	11	
AR (in %)	43.7	9.33	30	1		72.7	5.3	59.3	3.7	

Table A.4. Test set Lines
Goalie stands in yellow goal doing search-lines head-motion
On average, 30-33 line-points visible per image

LogFile	DT2004	Task
1 Flood	6	31
4 Floods	14	32
TL	19	34
TL+Floods	8	30
Tl+Floods+natural	14	30
Avrg (in %)	37	95.2

A.2. Using location information

A.2.1. Use distance information

We have used the OwnFlag() algorithm on two sets of log-files. The first set consists of images taken by the goalie guarding its goal. The second set consists of images taken by a player standing in the field facing the yellow goal.

Table A.5. Using distance information for goalie
Goalie stands in goal. The number detected flags is represented for when small flags are accepted (noSize) and when they are rejected (Size).

	NoSize		Size	
	Correct	False	Correct	False
searchLandMarks	7	2	7	0
searchlines	8	1	7	0
searchauto	13	1	10	0
lookstraight	0	2	0	0
playing	24	0	24	0
defend lighting	15	0	9	0
Total	67	6	57	0

Table A.6. Using distance information for player
Robot stands in field facing the yellow goal.

	NoSize		Size	
	Correct	False	Correct	False
centerfield	32	0	5	0
Goal1m50	8	1	7	0
goal3m	28	0	21	0
goal5m	17	0	2	0
Total	85	1	35	0

A.2.2. Disregard unexpected objects

Table A.7. Using location information for disregarding objects
The goalie stands in the centre of goal. The number of true and false positives is measured when a) only the own flags are searched for, and b): all flags and goals are searched for.

	OwnFlag		All Percepts	
	Correct	False	Correct	False
searchLandMarks	7	0	12	1
searchlines	7	0	11	3
searchauto	10	0	11	2
lookstraight	0	0	9	0
playing	24	0	26	1
defendlighting	9	0	15	3
Total	57	0	84	10

A.3.Processing power

The influence of the image processing algorithms on the frame-rate.

For several image processing solutions, we have executed the image processor algorithms N times. (1 < N< 100) and we have measured the frame-rate of the system. We have used a simple behavior where the robot used a *search-auto* head-motion and was standing still.

Table A.8. influence total vision system on frame-rate
For several image processor solutions, the number of executions is varied and the frame-rate was measured.

Impact of different vision solutions on overall frame-rate

	n Times Executed										
	n=1	2	3	4	5	6	8	10	15	20	25
DT2004	30	22	17	15	13	11	9	8	6	4	
Striker	30	23		15		12	10	8	6		
Position		22		14			8		4	3	2
lines+flag		30		29			21		15	13	10

Table A.9 influence of individual algorithms on the frame-rate.

	n Times Executed							
	n=5	10	20	25	40	50	70	100
Flag	30	28	22		17		11	
Goal	30	26		18		14		10
Lines	30	27		17		11		6
Lines(Hough)	9	4		1.5				

A.4. Real world tests

Table A.10: Localize in penalty area
Test time: 2 minutes.
Behavior: continuous-goalie-position head-motions: search-auto
By hand we place the robot in another place of the penalty area, facing the goal. Once the robot has returned to the centre of goal, the goalie is displaced again.
The number of successful returns is counted.

	Goalie	*Striker*	*DT2004*	*Lines + Flag*
1 Flood *	0	1	0	0
TL	11	5	0	0
3 Floods	14	8	6	5
TL + Floods	12	8	8	5
TL+Floods+Natural	12	8	10	9

*some real-time tests were performed at night, the pictures used for calibration were taken with some daylight coming through the curtains. Because of the darkness, the flags are not detected.

Table A.11: Return to goal

Test time: 3 minutes

Behavior for task goalie: playing-goalie-task. head-motion: unchanged

Behavior for striker and DT2004: cont-goalie-position head-motion: search-auto

The goalie is placed in centre of the field every time it has returned to the centre of own goal.

The number of determines the score.

	Goalie		*Striker*		*DT2004*	
	P1	P2	P1	P2	P1	P2
1 Flood	0	0	0	0	0	0
TL	6	4	6	4	0	0
3 Floods	8	4	6	4	2	1
TL+Floods	6	6	7	6	5	5
TL+Floods+Natural	5	7	6	5	6	3

Table A.12: Clear ball

Test time: 2 minutes

Behavior: playing-goalie-task head-motion: unchanged

The goalie is placed in centre of goal. The ball is placed at p1 or p2 when the goalie has returned to centre of the goal.

The number of ball clearances determines the score.

	Goalie	*Striker*	*DT2004*
1 Flood	5	0	0
TL	8	6	1
3 Floods	9	5	6
TL + floods	11	7	5

Table A.13: Clear ball, obstacles

Identical test as clear ball. Robots and other objects are placed in the field.

	Goalie	*Striker*	*DT2004*
TL	7	2	2
3 Floods	8	2	5
TL + floods	10	0	3