Team Research Report 2011

Nao-Team HTWK

Rico Tilgner, Thomas Reinhardt, Daniel Borkmann, Tobias Kalbitz,
Stefan Seering, Robert Fritzsche, Christoph Vitz,
Sandra Unger, Samuel Eckermann, Hannah Müller,
Manuel Bellersen,Martin Engel, Michael Wünsch

January 2012

Contents

1 Vision

1.1 Segmentation

The object recognition in Nao's camera image and identification of the field and objects on it is an essential part of playing soccer. The biggest problems for most color-table based methods are the inability to cope with changing light conditions and the need to generate the color-table, which can be very time consuming. Changing lighting conditions (e.g. between daylight and artificial light which is common at the German Open) make it impossible to classify objects solely based on their color. Also, differing ball colors (Robocup 2010) or unexpected carpet colors (Robocup 2011) pose another problem for purely color based methods. Therefore, a real-time capable segmentation with no need for calibration would be advantageous. By applying the knowledge of the objects' shapes we developed a object recognition algorithm that can handle changing light conditions and colors robustly without the need for prior calibration.

An in-depth description of the method is available in [1] (in German).

1.1.1 Program flow

Our object recognition is divided into several detection and filtering stages which are using the YCbCr color space. The order in which these stages are processed is of pivotal importance as each stage adds new information about the current image. These new pieces of knowledge are then used by later stages as references of the contents of the image. Examples for information which is passed on between the stages are distinctive color values of previously found objects and knowledge about their shape, position, size and rotation. The object recognition is done using the following five object properties which are not affected by the brightness of the objects in the picture:

property	example
difference in color	difference between the color of the ball and the color of the ground
shape	circular shape of the ball
size	the playing field's carpet is the biggest in size
position	the ball can only be within the bounds of the playing field
rotation	goalposts have a vertical orientation

Table 1.1: Object properties used during segmentation

However, a simultaneous use of all properties listed in table 1.1 is rare. Instead, each stage uses a selection of properties which is most suitable for its task.

1.1.2 The stages of the segmentation

First, the average color tone will be identified by finding the dominant color in the image, most likely the color of the playing field. In further steps, other objects can be distinguished from the field based on this information. All objects are complete or partially located in a specific area limited by the outer field lines. Finding these segments is part of the second step which decomposes the image into connected parts by examining the brightness channel. In a following step all line segments are assembled together to reveal the true game field lines and identify also possible false detections. The next step is designated to the search for the ball. Relying upon the previously collected data, field and line color as well as the circular shape of the ball, the its position can be determined. Knowing the position of the game ball makes the algorithm capable to obtain its color. The last and most challenging step is to find the goalposts. Once again, the program relys on the previously collected data which delivers all known color values. Goal detection is done by examining the Cb-Channel to match vertical edges above the detected field border which will finally lead to the position of the goalposts.

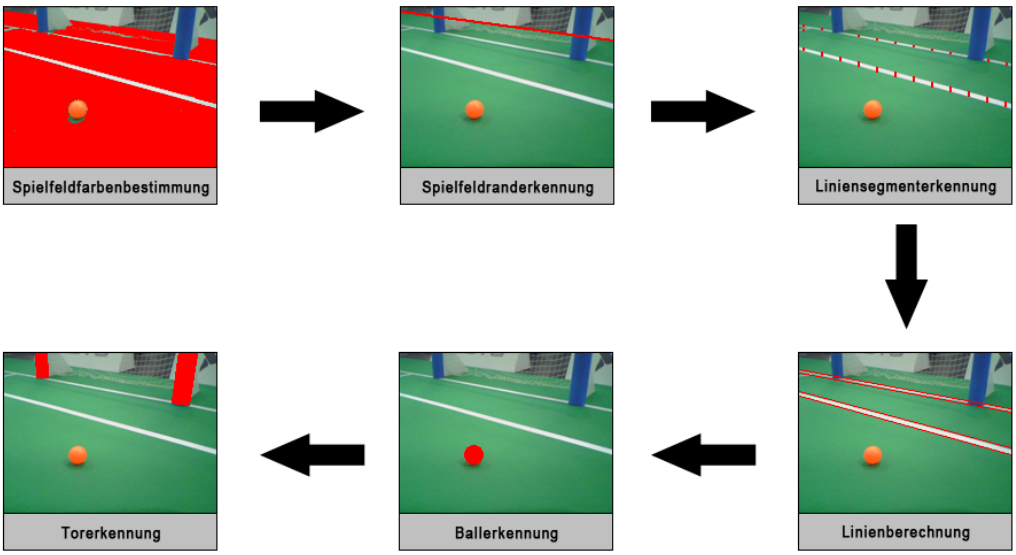

Figure 1.1: Schematics of the program flow during segmentation

1.1.3 Dataset for algorithm evaluation

For the evaluation of our image processing algorithms regarding detection rates we have implemented a database of testing images which can be found at [1]. This database contains about 600 images that we have captured with the robot's cameras. Each image contains a typical scenario found in games. Furthermore, we also included some specifially selected rare cases for evaluating the limits of our algorithms. During the

RoboCup German Open 2009 and 2010, as well as the RoboCup 2010 in Singapore, we captured those live images with our Nao robots. Based on this set of images that we have in our database, we developed our algorithms intending that also new game situations that were not covered by our images can be handled well. To gather a high diversity of game situations, we have captured those images from different positions, angles and in differing light conditions. The following table contains a frequency distribution of game objects we have captured in our database.

visible objects	frequency of occurence
ball (total)	40.2 %
ball (partially covered)	7.8 %
blue goal (both goalposts)	13.3 %
blue goal (single goalpost)	18.2 %
yellow goal (both goalposts)	11.3 %
yellow goal (single goalpost)	21.2 %
at least one visible line	89.0 %
no visible playing field	0.3 %

Table 1.2: Frequency distribution of game objects in our database

For every image in our database, the listed ground truth data from table 1.3 has been created manually. This data can then be used for an automatic quality evaluation of the developed algorithms.

object	data structure in database
field borders	Approximation by two lines
field colors	$(\bar{Y}, \bar{C}b, \bar{C}r)$-triple and information about visibility
field lines	several lists of coordinates that reside on edges of field lines
ball	coordinate of the center of the ball, radius and information about visibility
goals	coordinates and width of the post's bottom

Table 1.3: Stored data structure format of game objects of our database are presented in this table. The triple $(\bar{Y}, \bar{C}b, \bar{C}r)$ is defined as the arithmetic mean of the color values of all field pixles in a picture.

1.1.4 Results

detection of	hit rate	false-positive rate
field color	100.0%	0.3%
field borders	98.8%	1.5%
field lines	90.0%	4.6%
ball	99.2%	0.17%
goal	91.5%	9.0%

Table 1.4: Hit rates and flase-positive rates

Table 1.4 shows the hit rate and false-positive rate of the object recognition algorithm.

1.1.5 Program Runtime

function	runtime in ms			Number of accessed pixels in %		
	min	avg	max	min	avg	max
determination of field color	0.72	0.85	0.99	0.59	0.59	0.59
edge detection	5.8	6.3	7.6	3.1	3.4	4.1
detection of the field borders	0.50	0.82	1.7	0	0	0
line detection	0.31	4.8	7.3	0	0.23	0.85
ball detection	0.66	3.2	4.3	3.1	3.1	3.2
goal detection	2.4	2.7	2.9	0.83	0.83	0.84
total	10.4	18.7	24.8	7.62	8.15	9.58

Table 1.5: Runtimes and numbers of accessd pixels

Table 1.5 shows the runtime and memory access operations for the six main program parts measured during a five minute test game. All data are aggregated to its minimal, average and maximum values. The memory access operations to read the pixel values are displayed in relation to the whole image size; obtained from the test database and computed respectively. Owing to fact of other processes running in parallel to the segmentation, the minimal and maximal values are the median of 1 % of the true minimal and maximal values.

1.2 Localization

The localization method has been completely redesigned for the 2011 events.

The basic idea is to directly estimate the camera projection matrix from the segmented field-lines in the image instead of using the robot's kinematics or attitude sensors. This projection of field-lines is then used to find a complete set of hypotheses of the player's position, from which the true position can be determined by using prior data.

This method increases robustness of the localization in case of permanent camera movement (e.g. after a robot fell), fast head motions or external influences, e.g. in a fight with an opposing robot.

1.2.1 Estimating the projection matrix

Given the field of view and approximate height above the ground of a player's camera, we can determine an error function $f_{err}(\theta, \sigma)$ with roll θ and pitch σ of the camera according to the following postulates of projected field-lines:

- All field-lines (excluding the center-circle which is treated differently in the segmentation) are either orthogonal or parallel to each other:
  $f_{ang} = \sum_{i,j} min(|\angle(i,j) - \frac{\pi}{2}|, |\angle(i,j)|) \ for\, i,j \epsilon line_{\theta,\sigma}$
- The width of every field-line is 5 cm:
  $f_{width} = \sum_{i} width(i) \ for\, i \epsilon line_{\theta,\sigma}$

By weighting these two error functions in $f_{err} = w_{ang} f_{ang} + w_{width} f_{width}$ we can generate an error-function in θ-σ-space whose global minimum can be found reliably and with low computational cost using a simple hill-climbing algorithm.

1.2.2 Hypotheses Calculation

To compute all position hypotheses (x, y, α) on the field we use a brute-force method matching the projected field-lines and goal-posts to the known model of the playing field. As the lines are either parallel or orthogonal, there are only 4 possible directions α of the robot. The number of possible positions can further be reduced by only considering positions matching an intersection of the projected lines to an intersection in the model (i.e. either a corner or a T-junction). All remaining position candidates are then scored according to their conformance to the model, and implausible positions (e.g. having a 3 m long projected line where there only should be a 60 cm long line) are rejected. The remaining position candidates constitute the set of hypotheses which is used together with hypotheses from previous frames and odometry data to estimate the real position of the robot.

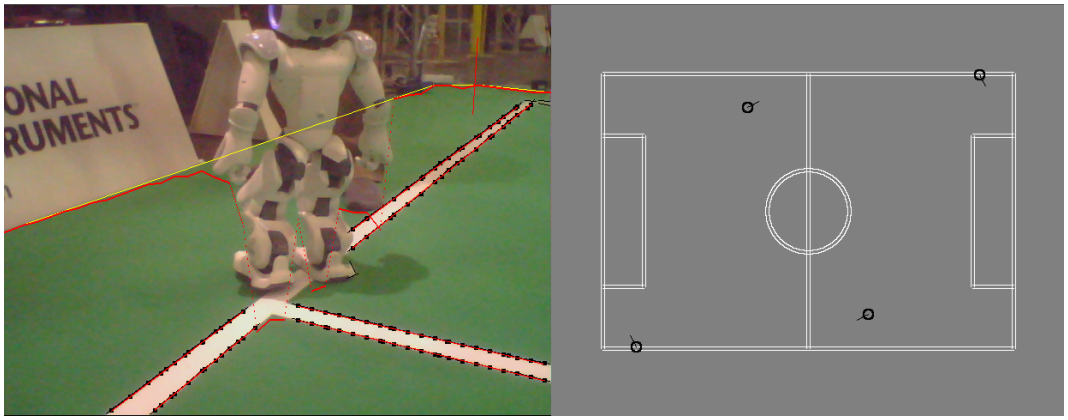

Figure 1.2: Segmented image and hypotheses calulated solely from this image

2 Motion

2.1 Walking engine

Until the beginning of 2010 we used closed-loop walking motions evolved through a genetic algorithm. These motions were fast but not omni-directional (eventhough walking along a curve was possible). This was a big disadvantage at the German Open 2010, so we decided to develop a completely different walking engine. Our walking engine, introduced in Robocup 2010 and refined in 2011 is based on a parameterizable walking model similar to [2] and is supported by a newly developed balancing algorithm. The big advantage of this system is full omni-directional capability and the ability to make fast direction changes whilst still being very stable.
The new walking engine was tuned for stability and speed manually and achieved forward speeds in excess of 300 mm/s.

2.2 Motion editor

The NaoMotionEditor is a replacement of Aldebaran's Choregraphe. The main purpose is to capture key frames directly from the robot, manipulate them and interpolate with a Groovy scripting engine between them. There exist already predefined Groovy scripts which define a linear and a smooth interpolation between two frames. These captured motions are saved in a XML file and can be later exported to a team dependent file format. To manipulate frames exist a variety of predefined operations like duplicate, mirror, move and show frames. The architecture of the editor is designed to add new functionality fast, so new requirements and features can be added on demand.

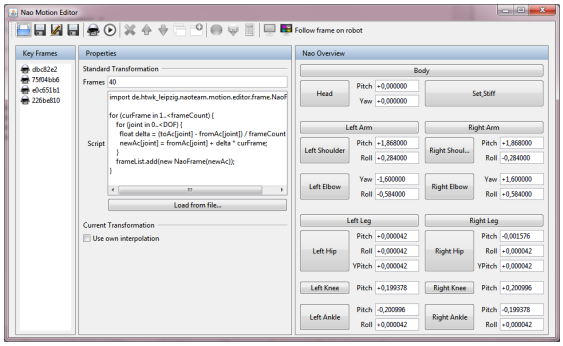

Figure 2.1: Example of a goalkeeper motion.

3 Framework

3.1 NIO

Our NIO (Nao Input Output) Framework is an independent piece of software that runs on Nao robots and extends the Aldebaran Robotics NaoQI framework.

The motivations for creating our own framework:

- Inconsistency of NaoQI's API
- Very limited debugging capabilities of NaoQI framework
- No need for a time intensive NaoQI restart after changes to parts of the software (e.g. motions, strategy)
- No thread safety of certain NaoQI calls
- Lots of NaoQI functionality we actually don't need
- Possibility of writing plain C instead of C++ code

The basic functionality of our NIO Framework is defined by a Unix Domain Socket client server pair. We have built a simple C++ module for the NaoQI framework that exports a subset of the NaoQI calls through the socket to the requesting process. This module is compiled as a shared library and will be linked against our actual kernel (core executable of our framework). Our exported API calls are kept very simple and performant, the subset is small and threadsafe. On the other hand, NIO consists of a series of subsystems. Each subsystem is generally independent of the others and serves one single aspect.

3.2 Memcpy improvements

Since memcpy seems to be a real bottleneck for video processing on Naos, we did some more investigation into optimizing its performance for larger chunks that need to be copied locally. AMD's V4L webcam driver for Geode seems to have serious driver issues with memory mapped I/O for video frames, so that we are forced to use Aldebarans NaoQI routines and copy video frames into a shared memory for NIO. We therefore measured the shipped memcpy function (libc) in combination with Geodes Time Stamp Counter processor registers (RDTSC) in order to gain current transfer rates. The investigation showed transfer rates of approximately 154 MB/s. Architecture dependent compile flags and other GCC optimization options did not result in significantly better

transfer rates. Since AMD's Geode has built-in MMX technology, we have developed our own userspace memcpy, which exploits the provided MMX instruction set. We could improve memcpy from 154 MB/s to 675 MB/s, which has an extensive impact in performance for frame transferal into our shared memory.

3.3 JVM communication and performace evaluation on AMD Geode

To improve development speed and lower the initial training of new team members we have researched and developed a proof of concept implementation of our NIO framework in Java.

Method	Time in ms
C-Server	0.1051
Oracle Java 1.6.20	0.1380
JamVM (no JIT)	0.7320

Table 3.1: Time for two TCP/IP function callbacks (1 TX, 1 RX).

Our tests in JVM and JIT speed indicated that Java is reasonably fast for strategy development and testing of prototype code. We plan to port our complete code base which is responsible for the strategy in the next few month to Java. The communication will be done by a TCP/IP connection for exchanging data from NaoQI to NIO and vice versa. Furthermore, we plan to generate the communication code layer by a program and evaluate the use of alternative communication channels like POSIX pipes and direct mapped memory which is also supported by JNI.

3.4 NaoControl

NaoControl is a monitoring program for our robots. It provides a virtual playing field showing the robot's and the ball's location. The Naos send their own and the ball's supposed position and an estimated localization quality to the program. With this, we can easily control whether the localization is fine or not. Also, the robot's rotation and field of vision is displayed.

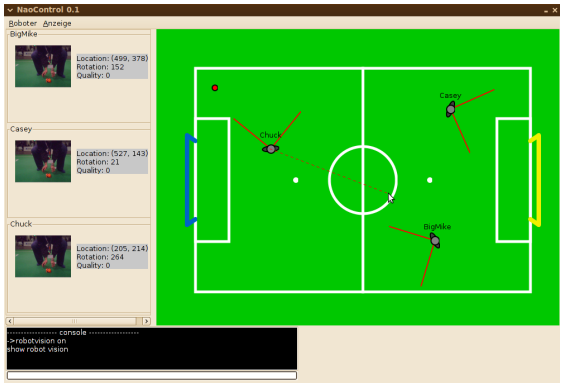

Figure 3.1: Screenshot of our NaoControl application.

Next to this, it is possible to show the actual images of the Naos' webcams. Those can be the real pictures or the segmented ones. We are able to send commands to the robots for testing them. The virtual playing field and its lines can be well customized, so new dimensions cause no problems. NaoControl is yet still in progress. In the near future it will be enhanced with simulation tasks. New playing strategies will be developed and tested with the assistance of NaoControl. For this purpose it provides simulated robot-behavior.

3.5 Networking - Naonet device driver

Our Nao robots do not communicate via IP-based traffic. Statically assigning IP addresses and netmasks can lead to misconfigurations during immediate competition setups of Naos or reboots of robots due to system failures. Therefore, we implemented a zero-configuration Layer 2 Ethernet protocol that runs within the Linux kernel by using the protocol hook *dev_add_pack* and Kernel FIFOs for queueing RX as well as TX packets. Communication happens via device files. A naonet communication manager within Userland controls the transmission and dispatch of messages. The receive path is therefore event-triggered. Several subsystems of NIO can register event-hooks provided as function callbacks to naonet's event manager. Depending on the specific event and destination (e.g. striker, defender, goalkeeper) all registered event-hooks will be triggered. The registration of hooks is priorized, so that high-priorized callbacks can stop the dispatch of messages for low-priorized callbacks. To sum up, we hereby have the possibility to notify other robots via event-driven mechanisms. This feature has shown robust and reliable communication in the Robocup events.

3.6 Wiimote control

We have developed a Wiimote remote-controlled Nao movement interface for several testing purposes. This enables us to play against our developed Nao game strategy or to efficiently test new motions, which for instance have been set up by our own motion editor. The Wiimote is conneted via Bluetooth socket to a Bluetooth-enabled host machine that is within the same subnet as the Nao that will be controlled remotely. On the Nao-side a server application listens for incoming instructions that are sent from the host machine. With the help of our software, the host machine can even route multiple Wiimote connections to various Nao robots.

Bibliography

[1] T. Reinhardt, "Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball," Master's thesis, HTWK Leipzig, 2011. [Online]. Available: http://thomas-reinhardt.de/Vision/

[2] S. Behnke, "Online trajectory generation for omnidirectional biped walking," *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 1597 – 1603, May 2006.