Master's Thesis

Autonomous Nine Men's Morris Position and Game State Recognition using a Humanoid Robot Platform

Sven Bock

Graz, 2012

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Dr. techn. Gerald Steinbauer

Abstract (English)

The motivation of this thesis is to teach humanoid robots to play board games. As our board game of choice, we selected Nine Men's Morris. The game has a simple setup, but due to the possible combinations of the game tokens on the board, it has a high complexity level. The humanoid robot "Nao" has been chosen as the robot to interact in the world on a common sized game board of Nine Men's Morris, mounted on a customized table for the robot. As the robot is too little to manipulate all objects on the table from one side, it needs to walk around the board, which requires an accurate self-localization. In order to support the localization, an augmented reality marker (AR marker) has been used. All vision tasks are accomplished, only using images of the two internal cameras of the robot. The computation of a move in the game requires the current game state. Therefore all game tokens on the board have to be recognized and correctly classified. The best next states are saved in a database. After the robot has computed the next move out of the state, it has to execute it. Hence it moves its torso to a position in front of the board, that can reach the target three-dimensional token position with the fingers of one arbitrary arm. A grasp with the fingers for a token always needs to be confirmed with a camera image, because the fingers of the robot do not have touch sensors. In case a grasp fails for the first time, the robot has some repair procedures to fix the task on its own, if it is possible. Otherwise human help is requested by the robot.

The complexity of the task to play a board game requires a modularization in smaller parts, whereas the focus of this thesis lies on the image processing, the game logic and a framework for overall control. The game-play framework is modularized in the following three categories:

- **image processing:** Recognition of the game board and all tokens, self-localization with the help of an AR marker and visual support during walk procedures with one of the built-in cameras; Fusing of game board observations and different sensor sources and calculation of efficient strategic movements;
- **mathematics:** Computation of the next best step based on the current game state realized by a precomputed database;
- **kinematics:** Computation of all joint angles and timed paths to successfully place and grasp a token. Additionally a planner to walk around the board is required;

We faced some challenges that arose when different technologies were integrated. These had to be solved in order to proceed. Even if some approaches of this work already existed, the complete integration of all parts to play Nine Men's Morris autonomously makes this work unique at this moment to the best of our knowledge. In order to solve the task special methods have been adapted during the implementation, for example the robust token classification, the visual odometry, efficient strategic walk and grasp planning and a specific kinematics library for our humanoid robot. The kinematics library, as well as the walk and grasp planner, which are used, were developed by Klöbl [28].

Abstract (German)

Die Motivation dieser Arbeit liegt darin, menschenähnlichen Robotern einfache Brettspiele beizubringen. Als Brettspiel wurde in dieser Arbeit das Spiel Mühle verwendet. Trotz seines einfachen Aufbaus besitzt dieses Spiel, durch seine Vielzahl an möglichen Spielsteinkombinationen, einen hohen Komplexitätsgrad. Als humanoider Roboter wurde "Nao" ausgewählt, um mit der Welt zu interagieren. Für diesen Roboter wurde ein gewöhnliches Mühlefeld auf einen speziell für den Roboter angefertigten Tisch plaziert. Da der Roboter zu klein ist um alle Spielsteinpositionen von einer Tischseite zu erreichen, muss er um das Spielfeld herumgehen, was eine genaue Selbstlokalisierung vorraussetzt. Zur Unterstützung der Selbstlokalisierung wurde ein Augmented Reality marker (AR marker) verwendet. Alle visuellen Aufgaben, wurden erfolgreich nur mit den zwei internen Kameras gelöst. Die Berechung des nächsten Spielzuges erfordert den aktuellen Spielstatus. Dafür müssen alle Spielsteine auf dem Spielbrett erkannt und richtig zugeordnet werden. Die besten möglichen Folgestellungen sind in einer Datenbank gespeichert. Nachdem der Roboter aus der Spielstellung seinen Zug errechnet hat, nimmt er mit seinem Oberkörper eine Position vor dem Spielbrett ein, in welcher einer von beiden Armen die dreidimensionale Zielposition greifen kann. Der Roboter muss die Greifbewegung für einen Spielstein immer visuell mit dem aktuellen Kamerabild bestätigen, da in den Fingern keine Berührungssensoren vorhanden sind. Sollte ein Griff beim ersten Mal fehlschlagen, hat der Roboter Möglichkeiten seinen Fehler selbst zu korrigieren, sofern das möglich ist. Ansonsten fordert der Roboter menschliche Hilfe an.

Aufgrund der Komplexität der gestellten Aufgabe teilt sich dieses Projekt in mehrere Einzelteile, wobei in dieser Arbeit der Fokus auf der Bildverarbeitung, der Spiellogik und dem Framework liegt. Das Projekt teilt sich in folgende Disziplinen auf:

- **Bildverarbeitung:** Erkennung des Spielfeldes inclusive aller Spielsteine, Selbstlokalisierung mit Hilfe eines AR markers und visuelle Unterstützung bei Bewegungen mit der eingebauten Roboterkamera
- **Mathematik:** Berechnung der nächsten Züge aufgrund der aktuellen Spielsituation durch eine vorberechnete Datenbank; Fusionierung der Spielfeldbeobachtungen und Sensorquellen; Effiziente strategische Plannung
- **Kinematik:** Berechnung der Gelenkswinkel und Zeitintervalle der zum Plazieren und Greifen vorgegebenen Spielsteinposition; Zusätzlich werden Planner benötigt, welche Gehbewegungen um das Spielfeld herum planen und eine Position zum erfolgreichen Greifen berechnen und ausführen

Bei der Integration verschiedener Technologien traten immer wieder unvorhergesehene Herausforderungen auf. Diese mussten erst gelöst werden, bevor die Weiterentwicklung fortgesetzt werden konnte. Auch wenn einzelne Ansätze dieser Arbeit bereits existieren, macht die komplette Integration aller Teilbereiche, um autonom Mühle richtig zu spielen, diese Arbeit unseres Wissens nach bisher einzigartig. Um die gesamte Aufgabe zu lösen wurden einige Methoden benötigt und adaptiert, wie zum Beispiel die robuste Spielsteinklassifikation die visuelle Odometry, eine effizientе strategische Planung, sowie Planer zum Greifen und Gehen und eine eigene Kinematikbibliothek. Die Herausforderungen zum Gehen und Greifen, sowie die Kinematik wurden von Klöbl [28] erarbeitet und gelöst .

Acknowledgements

I would like to thank my parents, Holger and Iris, who always supported me with everything I needed. I also want to thank my brother Björn, who was nice enough to correct parts of this masters thesis and encouraged me to seek my own path. Moreover I would like to thank my girlfriend Anna, who gave me confidence in myself and paid attention when I was talking about project specific topics. Furthermore I would like to express my gratitude to my grandparents, who sponsored my study and Roland, my colleague, who developed a kinematics for the robot and helped me to find errors. I also want to thank my good friend Christian, for our frequent discussions, where new ideas where born to solve challenging tasks. Last but not least I want to thank Gerald Steinbauer, who presented an autonomous robot on the open house day of the university in 2005 and thus convinced me to study telematics. He always was friendly and supported me, whenever I needed help. Thanks to Gerald Steinbauer for the interesting topic which was a lot of fun to develop!

Sven Bock
Graz, 2012

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen, wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Contents

List of Figures vii

List of Tables ix

Listings ix

1 Introduction 1
- 1.1 Motivation for the Project . . . 1
- 1.2 Goal . . . 2
- 1.3 Project Overview . . . 2
- 1.4 Definitions and Terms . . . 3
- 1.5 Presentation of the Problem . . . 6
  - 1.5.1 Challenges . . . 8

2 Sensors, Techniques, Hardware and Environment 10
- 2.1 The Robot . . . 10
- 2.2 NaoQi, Python, C++ and the game engine . . . 13
- 2.3 The Game Board . . . 14
- 2.4 The Marker . . . 14
- 2.5 Game Tokens . . . 15
- 2.6 Hardware - Processing Notebook . . . 15

3 Related Work - Research 16

4 Concept to solve the Task 22
- 4.1 Overview . . . 22
- 4.2 Basic Nine Men's Morris Rules . . . 22
- 4.3 Development Environment . . . 23
- 4.4 Basic Concept . . . 24
- 4.5 Framework architecture and Features . . . 26
  - 4.5.1 Package: high level control . . . 27
  - 4.5.2 Package: planners . . . 27
  - 4.5.3 Package: game database . . . 28
  - 4.5.4 Package: world model . . . 28
  - 4.5.5 Package: helpers . . . 28
  - 4.5.6 Package: motion . . . 29
  - 4.5.7 Package: vision . . . 29

5 Solution to the Task 30

5.1 High Level Control Package . . . 31
5.2 Planner Package . . . 32
5.2.1 Motion Planner . . . 32
5.2.2 Analyze Field . . . 34
5.2.3 Color Calibration . . . 34
5.2.4 Kinematics . . . 35
5.2.5 Movement and Path . . . 35
5.2.6 Walk Planner . . . 35
5.2.7 Grasp Planner . . . 36
5.2.8 Movement Behaviors . . . 37
5.2.9 Look At . . . 37
5.3 World Model Package . . . 38
5.3.1 World Model Class . . . 39
5.3.2 Game Board Representation . . . 39
5.3.3 Game Board Class . . . 40
5.3.4 Transformations and Localization . . . 42
5.3.5 NaoTactile Class . . . 47
5.4 Database Package . . . 47
5.4.1 Database Simulator . . . 49
5.4.2 Cheat Detection . . . 49
5.5 Helper Package . . . 49
5.5.1 Interface: NaoQi - ROS . . . 50
5.5.2 Definitions . . . 51
5.5.3 Helper Functions . . . 51
5.6 Vision Package . . . 51
5.6.1 Camera Module - Image acquisition . . . 52
5.6.2 AR ToolKit . . . 53
5.6.3 Image Preprocessing . . . 55
5.6.4 Estimation of Distances and 3D Points . . . 57
5.6.5 Projection of the 3D tokens into the Image Plane . . . 60
5.6.6 Blob Classification . . . 62
5.6.7 BW Label . . . 68
5.6.8 Visual Odometry . . . 68
5.7 Motion Package . . . 70
5.8 Game Tokens . . . 70
5.8.1 Production of the Game Tokens . . . 72
5.9 Other Packages . . . 73

6 Experimental Evaluation **74**
6.1 Network Traffic . . . 74
6.2 Token Placement . . . 74
6.3 Token Grasp . . . 76
6.4 Token Classification . . . 78
6.5 Walk and Localization . . . 81

7 Conclusion **83**
7.1 Network Traffic . . . 83
7.2 Place . . . 84
7.3 Grab . . . 84
7.4 Get Game State . . . 85
7.5 Walk . . . 85
7.6 Future Work . . . 86

Bibliography **87**

List of Figures

1.1 The marker situated in the middle of the game board . . . 3
1.2 The complete game board including the marker . . . 3
1.3 The real game board from above with strategic positions, sectors and marker . . . 4
1.4 All masks combined in one image . . . 5
1.5 The robot in its initial position next to the table . . . 7

2.1 Overview of Nao's features . . . 10
2.2 Camera module showing the two cameras . . . 12
2.3 Head of Nao with the position and orientation of both cameras . . . 12
2.4 Touch sensors on the head module. . . . 13
2.5 Two sample tokens of yellow and purple color . . . 15

3.1 Machine-like and human-like robot . . . 21

4.1 Final game tokens for both players . . . 23
4.2 Model of the communication between the packages of the framework and the robot . . . 26

5.1 Dependencies of nodes, waiting for another . . . 30
5.2 High level control tasks. . . . 31
5.3 Control flow of the overall state machine . . . 31
5.4 Planner tasks . . . 32
5.5 Procedure to repair a place action . . . 34
5.6 Path around the board . . . 36
5.7 Procedure to find the marker . . . 38
5.8 World model tasks . . . 39
5.9 The transformation tree of all coordinate systems . . . 43
5.10 The transformation tree of the robot containing legs, arms, head and cameras . . . 43
5.11 Representation of different coordinate systems . . . 45
5.12 Database tasks . . . 47
5.13 Helper tasks . . . 49
5.14 Vision tasks . . . 52
5.15 Marker detection in the camera image . . . 54
5.16 External view, while the robot detects the marker . . . 54
5.17 Final color detection, colored and automatically drawn in a camera image . . . 56
5.18 Images for the detection for tokens in the hands . . . 57
5.19 Draft of the focus point on the board . . . 58
5.20 Draft of the blue stripe distance calculation . . . 59
5.21 Projection of three-dimensional point of the scene into the image plane of the camera . . . 60
5.22 Calibration target to obtain the intrinsic camera parameters . . . 62

5.23 Example of multiple tokens detected as one blob . . . 63
5.24 Example of two separate detected blobs in one mask . . . 64
5.25 The plane mapping of a projection . . . 65
5.26 Sidelengths of combined mask and real game board . . . 66
5.27 Focus points . . . 67
5.28 Sample of two images after rotation with visualized matches. . . . 69
5.29 Difference angle computed of two similar camera images . . . 69
5.30 Motion tasks . . . 70
5.31 A HSV cone that shows all channels . . . 70
5.32 Histogram of game tokens and blue stripe . . . 71
5.33 The pincer fingers and the corresponding wirerope to grasp something . . . 72
5.34 Alum power in solid form . . . 73

6.1 Pre-placement position of the hand . . . 75
6.2 Placement on the middle square . . . 75
6.3 A grasp of a token on the inner square . . . 76
6.4 Grasping procedure on a middle position . . . 77
6.5 Different token poses inside the fingers . . . 77
6.6 External view to a possible game state during a game . . . 78
6.7 External view to an artificial game state setup . . . 79
6.8 Game state detection result . . . 79
6.9 Game state representation in rviz . . . 80
6.10 Walk around the board . . . 81
6.11 Visual odometry output . . . 82
6.12 Visual odmetry output . . . 82

7.1 A misplaced token on the inner square . . . 84
7.2 Unsafe grasp with two fingers . . . 85

List of Tables

2.1 Tracking ranges for different sized patterns 14

5.1 Conversion table of game state elements 48
5.2 Game phase encoding 48
5.3 Edit option encoding 48

6.1 Average network traffic between the NaoQi and ROS 74
6.2 Evaluation of token placements 76
6.3 Evaluation of token grasps 76
6.4 Evaluation of token and game state classification 78
6.5 Evaluation of walks around the board 82

Listings

5.1 ”Pseudo code for initial color calibration procedure” 35
5.2 ”Example code to import the NaoQi library and Python for ROS.” 50
5.3 ”Example code to get an image from the Nao camera” 52

List of Algorithms

1 Calculation of best destination sector and arm 41
2 Computation of the eight points for rectification 67

Chapter 1

Introduction

Recently the number of autonomous robots has grown strongly, Although they are not obviously visible, they are not restricted to a certain domain. Many people imagine an autonomous robot with two arms, two legs and a head. This physical appearance is called a humanoid robot. The robot in a milking plant that autonomously milks the cows in a stable, can be count as autonomous as a humanoid soccer robot or any vacuum cleaner bot.

Personally I believe that autonomous robots will be very important in future households, warehouse management and other domains. I think there are many people who would appreciatively accept a humanoid household robot, which makes their life more comfortable, like many other electronic tools in the kitchen already do. In the science fiction movie ”I-Robot” of 2004 for instance, there are humanoid robots with very advanced artificial intelligence to help humans in their everyday life. It will take some time until our current robots are at the same state as the robots in science fiction movies are now - if it is even possible.

By now the data storage devices like hard disk drives and processing units are far too limited to store and simulate a complex mind. Even a simple single mathematical challenge like the game ”connect-4” holds too much possibilities to store every possible step. We would need 1000-times bigger hard disks just for that task. So storage is still an issue, if we want to model complex efficient environments.

Someday it should be possible to buy a humanoid robot for a few hundred euros, which is able to play some traditional board games at home with children or other human beings who like real board games. In this master thesis I want to present my work on a humanoid robot that is capable of playing Nine Men’s Morris autonomously. It is only a little contribution, but certainly one step to the future. At first I wanted to develop everything on my own but soon I noticed that the variety of different domains in which I need to create something is huge, so the project of playing Nine Men’s Morris with Nao was split in several projects.

1.1 Motivation for the Project

”Why can’t a humanoid robot play a typical board game with me?”

This was my question and a desire of mine that was in my mind at the start of the project. Sometimes I want to play a board game even if none of my friends are present. ”Why can’t a robot replace my play mate, at least while a board game?” I would rather want to see my future children playing with a robot in the real world than playing a game on the computer. I think it is important for the development of children to navigate in a three-dimensional world to improve their imagination, their stereoscopic vision

and movement. Personally I think this is a possibility to keep children away from playing computer games for some time, similar to remote controlled vehicles.

The question of playing board games autonomously leads to many other questions that have to be answered previously. Autonomous actions in the real world for a humanoid robot are very complex. Of course there are solutions to individual tasks, but only a few cover a complete strategy to play a board game. The present hardware technology is able to perform this task, but the intelligence of a robot still is a challenge. Mankind is able to create artificial proteases and organs, but the human brain is still a mystery for us. Many years ago I read a quote by Gaarder [19] that said, "If the human brain was simple enough for us to understand, we would still be so stupid that we couldn't understand it." This means that we are never able to understand our own brain completely and we may only create an artificial intelligence that is inferior to our own intelligence. Although I like this way of thinking, I believe it is limiting oneself. Nevertheless I think autonomous robots are an important contribution to our civilization. They may be always inferior to ourselves but in special tasks they really could support us. I am excited to see how intelligent humanoid robots are accepted by mankind in future.

1.2 Goal

Due to the progress in technology, I want to show that it is already possible to play a typical board game with a humanoid robot. The humanoid robot platform, Nao, was built by the French company Aldebaran. It is used to play the board game Nine Men's Morris in this project. For this approach the robot should be totally autonomous. This means that the robot has no external sensor information about the board and does not receive any human input for its actions. The robot recognizes the board, evaluates the next step and proceeds with the plan or repairs it, if it is possible. This project is combined with a completely solved Nine Men's Morris database from another group of the university. The database requires approximately 100 GB of free disk space. I am convinced that it will be possible in the future to use this kind of software on the robot itself. Maybe the database can be even more compressed by not storing all possible combinations. I think if the program is mature enough for simple and stable usage, it is ready for commercial purposes. The robot could be equipped with different memory sticks, each storing the specific data, required for its specific task, like the board game Nine Men's Morris. These sticks could be swapped like game cassettes in a traditional "Nintendo GameBoy". If the production of the robot and the demand is high enough, such a robot should be affordable for every average household.

1.3 Project Overview

This section provides an overview, as well as a brief description of all chapters.
Chapter 2 introduces the used hardware, the robot and the notebook, the used sensors of the robot and the environment consisting of the game board table, the tokens and the marker. Furthermore the used framework and programming languages are explained briefly.
Chapter 3 gives an overview of related projects and different board games with humanoid robots. Localization with a single camera, efficient segmentation, vision-based grasping and artificial gaming intelligence are being discussed among other related topics.
In Chapter 4 the basic rules of the game, as well as the main concept and framework structure to solve the task are explained.
The solution to the task is explained in Chapter 5, where vision tasks are focused strongly and the framework structure is explained.
Chapter 6 contains the experimental results of different high level actions, such as grasp, place, walk and the token classification.
In Chapter 7 the results of the experiments are discussed.

Our framework is open source and available for download at [8].

1.4 Definitions and Terms

In this section all terms, definitions and abbreviations for this specific project are explained briefly.

AR marker is the abbreviation for ”Augmented Reality” marker. It refers to a special known black and white square in one plane. The complete pose of the camera or marker, can be estimated with the known marker structure. Therefore it is used for localization and mounted in the middle of the game board to be equally visible from each side. The used marker is shown in Figure 1.1.

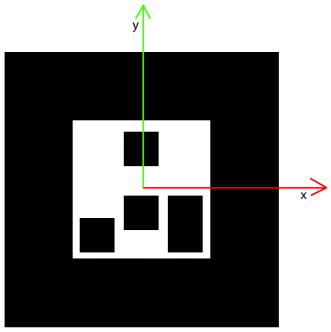


Figure 1.1: The marker is situated in the middle of the game board. The non-symmetric pattern is essential for a reliable orientation information. The thick black objects improve the recognition and the black-white coloring enhances the contrast to a maximum. The x-y marker system is also shown. The z-axis is the vertical axis coming out of the image.

Blue stripe is the colored tape on the border of the board, which is used for course localization (see Figure 1.2).

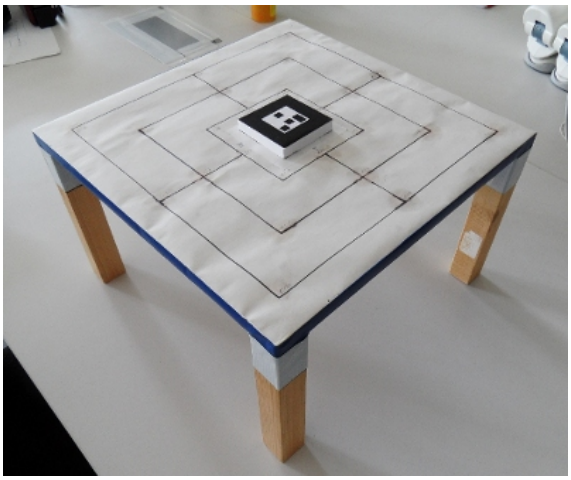


Figure 1.2: The complete game board including the marker

Square is always related to the four lines that surround the marker of the game board in an equi-perpendicular distance. The board is divided in three black, non-filled squares, as can be seen in Figure 1.3. The inner square is defined by the four lines that are closest to the marker. The outer square is defined by the most distant four lines and the middle square consists of the four lines in between.

Element is a point of an intersection on one of the lines of a square (see Figure 1.3). The element number increases counter clockwise from zero to seven, which results in eight elements for each square.

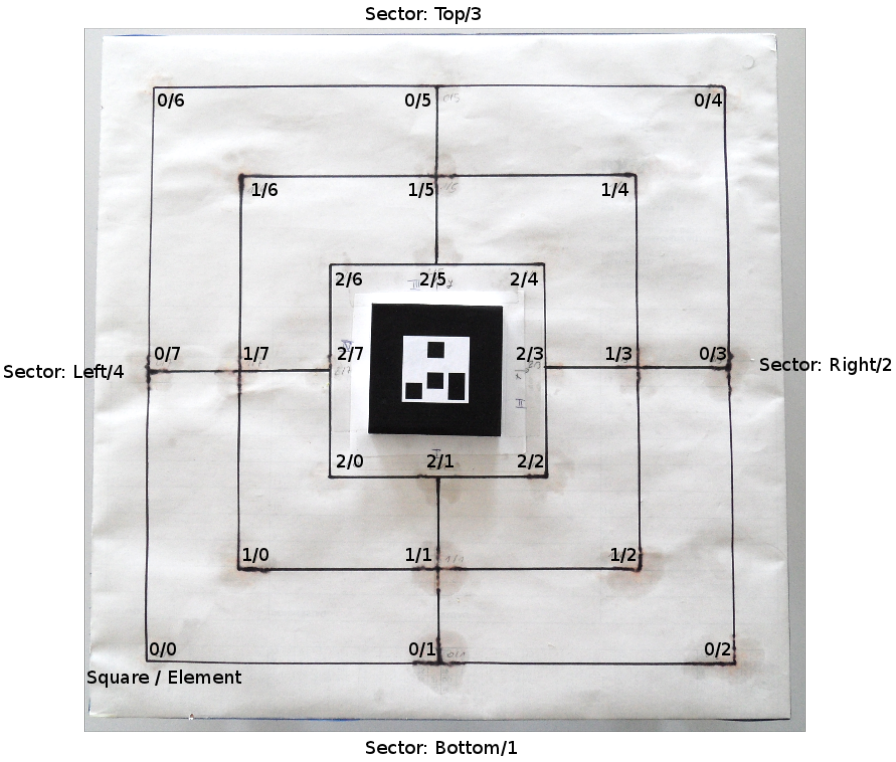


Figure 1.3: The game board from above with strategic positions, sectors and marker. The first value of each position represents the square number, while the second one represents the element number.

Strategic positions are abstract positions of the game board. All of the 24 positions contain the occupation to represent the game state. One position is addressed by a square and element or an index. There are three squares and eight elements, while the index holds 24 entries.

Ideal positions are the positions of the game board, where all game tokens should be. Every intersection of the lines on the board is an ideal position (see Figure 1.3).

Real positions are the estimated positions of the tokens centers, where they really are on the game board. Due to different token sizes and imprecise placement, tokens never lie on their ideal position.

Sectors are specific two-dimensional areas around the game board, used for strategic decisions. Referring to Figure 1.3, the bottom side is defined as sector ”0” or ”BOTTOM”, the right side as ”1” or ”RIGHT”, the top side as ”2” or ”TOP” and the left side as ”3” or ”LEFT”. The four sectors are divided by two lines that intersect in the center of the marker and game board. The marker system is shown in Figure 1.1. One line goes through the strategic elements 0 and 4 with 45° to the x-axis of the marker and the second line goes through the strategic elements 2 and 6 with 45° to the y-axis of the marker.

Point/Position refers to a two or three-dimensional position, for example (x,y,z).

Pose refers to a six-dimensional or seven-dimensional position, like a point with an orientation, for example (x,y,z, ϕ, ψ, θ)

Mask is an area on the board that belongs to a position. Due to the fact that the masks are not overlapping each other, all 24 masks can be visualized in one combined image, which can be seen in Figure 1.4. Each white connected area represents one mask.

Visual odometry refers to a service that supports the localization with the help of camera images, if the robot rotates.

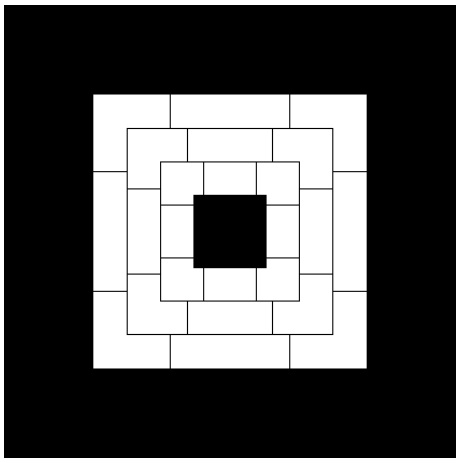


Figure 1.4: All masks are combined in one image. Each mask belongs to a strategic position and is divided by a black border to other masks. Every pixel may be clearly classified to one mask. The black border is used to cope with border conditions.

Blob originates from "Binary Large OBject" and in this project it denotes a homogeneous circular shaped area that differs from the background and represents a token.

Multi blobs are present if two or more blobs are so close to each other that they cannot be separated. They melt to one blob that no longer has a circular shape. Multi blobs are more difficult to classify. Their three-dimensional real position is also difficult to estimate.

Principal point offset refers to the pixel distance in a camera from the middle of the light-sensitive chip to the optical axis.

NaoQi is the framework running on the robot to control the hardware. There is a rough documentation on how to use the NaoQi [4].

Game phase refers to the current progress in the game. The main game phases are explained in Section 4.2.

Place is the term of the procedure to put a token from the storage on the board during the first game phase.

Take is the term of the procedure to grasp or remove a token completely out of the game.

Move is the term of a manipulation of a an already existing token on the board. It includes the grasp and release on another valid position on the board.

Denavit-Hartenberg-Transformation is a mathematical standard procedure to describe kinematic chains to handle coordinate transformation.

Potential field is a method that is used in path planning. The algorithm is only allowed to move from a higher to a lower potential on a map, while obstacles are initialized with a very high potential.

RGB is the most common cuboid shaped color space. It combines every color by an additive combination of the three basic colors *red*, *green* and *blue*.

HSV is the most common cylindrical shaped color space. Instead of using the three colors it splits the final color in *hue*, *saturation* and *value*. *Hue* represents the color or wavelength. *Saturation* defines more or less how pure the color is. The *value* corresponds to the brightness or intensity of energy.

YUV422 is similar to the HSV color space that encodes the brightness in an extra channel, the Y-channel. YUV422 [44] is a special encoding that encodes 2×3 bits in 4 bits.

Median is the central value of a list of numbers. Most of the time the list gets sorted in ascending order before extracting the central value. This procedure is used to find a good mean value ignoring outliers.

Transformation/tf refers to a coordinate transformation, where the coordinates of one system are converted into another system. A transformation matrix can be established to easily transform points by a multiplication. In the typical robotic domain this transformation matrix contains translations and rotations only.

Homography is a projective transformation, which it is used for vision tasks. In this project the homogeneous matrices for the camera images are calculated to rectify them. The projective transformation is only applied in the two-dimensional space.

SURF stands for ”Speeded Up Robust Features” [7] and is an scale and rotation invariant keypoint descriptor with very good repeatability, distinctiveness and robustness.

ROS is the abbreviation of “Robot Operating System” and provides libraries and tools to help software developers to create robot applications [36];

URDF model refers to ”Unified Robot Description Format”, which is an XML format for representing a robot model [36].

Rviz is a stack of [36], used to visualize data. It handles camera images, transformations and many geometric shaped objects to visualize an abstract world.

Quaternions are a complex number system to extend the complex numbers. They are used for three-dimensional navigation as they always perform an known trajectory [43].

1.5 Presentation of the Problem

On start up the robot is sitting next to its Nine Men’s Morris game board, oriented towards the game board as shown in Figure 1.5.

Whereas every human player would check, where the game board and its tokens are, the robot initializes itself. Initialization for the robot means standing up and finding the marker. This first step is very important, because it localizes the robot in the world. Afterwards the robot needs to know the token colors of both Nine Men’s Morris players to correctly classify them during the game. Additionally the robot initializes a blue color to find the blue stripe on the border of the board which helps to relocalize the board from a distant position. There is a color calibration method to automatically detect the three colors. As the last procedure of the initialization step, it sets the game board occupation. On a new game all game board positions are initialized as empty. However there is an option to continue a game. In this case instead of a free board the robot scans the whole board to retrieve the current game state.

Due to a small opening angle and a close distance to the board, only a few positions can be seen at a time. Therefore the procedure of retrieving the game state is a complex function that includes head motion, coordinate transformation, projection of three-dimensional points, rectification, matching and validation of the observation.

After the initialization the robot and the human player place tokens one by one. Before the robot may place a token, it has a lot of things to do. Firstly, it has to self-localize using the marker and acquire the game state. Secondly, it can place or move and even take a token. It may only take a token, if it created a mill which refers to three tokens in a line. The robot uses an internal database that always computes its next best move to the current state. Before the new state is requested, the current state is validated to detect probable cheats or game inconsistencies.

The execution of the move is the next task. The robot always selects the shortest way to go. The selection of the arm corresponds to the shortest walking distance, because a movement to another table side is extremely time exhaustive. While the robot walks to its new table side, the visual odometry supports the rotation around the robot’s vertical axis by measuring the rotation angle.

When the robot arrived at its new table side and relocated itself, the target token position is refined if it is a ”take” or ”move” action. A ”place” does only require the ideal position, which is always known

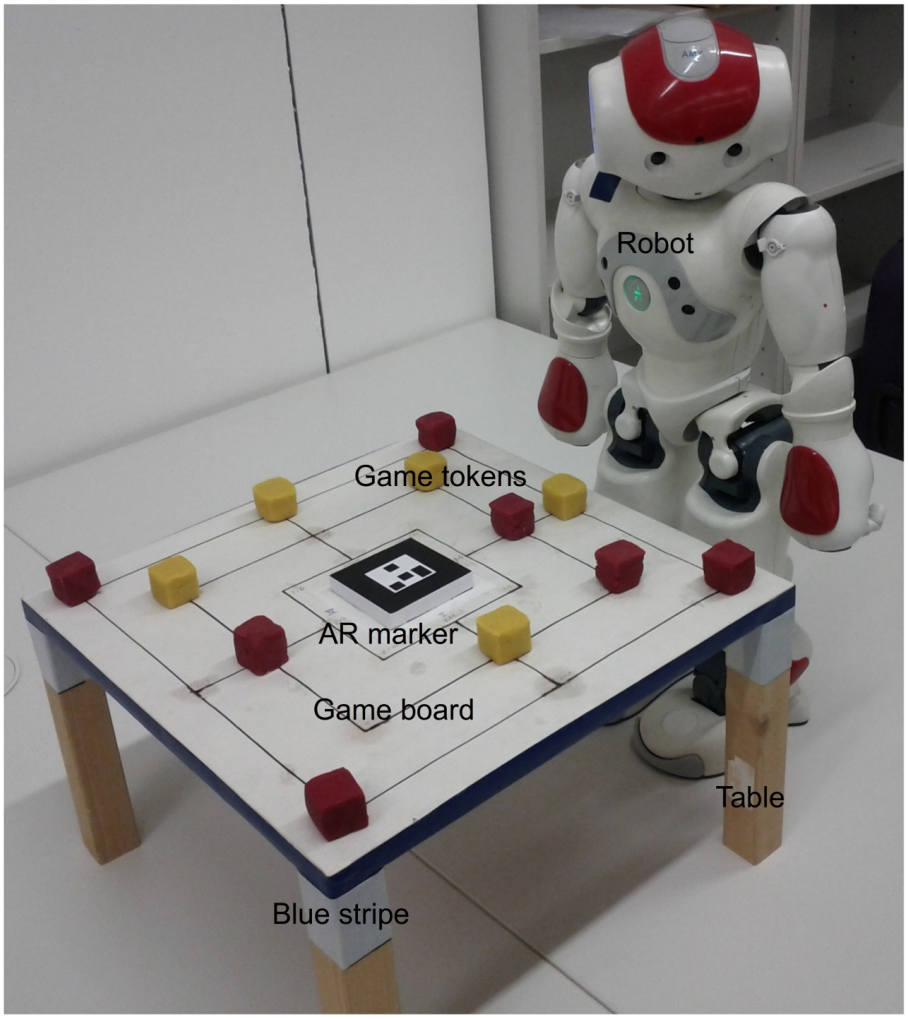


Figure 1.5: The robot in its initial position next to the table. Any position like this around the table serves well as initial position.

in the world model. Refining means to look directly at the token at close distance to minimize its real token position error. Then the kinematic computes all angles of the robot to reach the target position and the grasp-planner tries to find a valid ground position for the robot where it needs to be in order to manipulate the token. If the token is not graspable while standing in the current position, the robot corrects itself and manipulates the token. Afterwards it checks if the manipulation has succeeded by looking at the target position. If the vision confirm that the manipulation has failed, there are different repair procedures, depending on the manipulation mode. For instance if a ”place” has failed, then it is most obvious that the token rolled away or tilted. In this case the robot has to search for the lost token and place it again. In the worst case the token has fallen off the table and is unreachable for the robot, so that it needs human assistance. In case the token manipulation worked as expected or was successfully repaired, the robot takes a resting pose until the human player finished a turn. Nao takes its next turn when the human player touches its front head. This was one turn of Nine Men’s Morris from the point of view of a humanoid robot.

1.5.1 Challenges

Game state recognition is a challenging task as it requires the exact localization of the robot and the game tokens. The task to calculate the token positions would be much easier with a global camera above the game board, because the image would only be distorted minimally. However we chose not to use external sensors, because they require a specific setup environment, which need to be calibrated. The idea to use only internal sensors and a game board marker, which can be easily printed on every printer, qualifies each Aldebaran Nao robot to play Nine Men's Morris. The robot just needs the specific Nine Men's Morris gaming software and a printout of the game board, as well as a table of a specific size. Some tokens that are graspable by the robot are also required, but they would also be needed if a global overhead camera was used.

One goal was to provide a good reuseability of our framework, because it should be able contribute to future implementations of gaming robots. This implies a slim and robust framework that is easy to install and working on a majority of computers. The framework should be split into modules of different tasks, so that each module can be replaced in case of improvements or adapted usage. In case any fixed sizes of our project are different, the user can easily adjust the parameters in one file before starting the framework.

Another challenge are the inconsistencies of the game board tokens. What happens if there are more than one token of different colors in a mask? Is this valid? What can we do?

Due to the decision to use internal sensors only to detect the Nine Men's Morris game state, the localization is restricted to the odometry and the internal RGB camera. The internal odometry of arms, head and legs in relation to the robot itself is much more precise than the estimated pose of the robot in relation to the external world. The latter is drifting away very fast from the real pose due to the accumulation of errors. The odometry value of the angle around the own vertical robot axis is the most inaccurate value. One of the reasons may be a missing gyrometer in the horizontal plane. Moreover it is also nearly impossible to make small movements or rotations (for example a forward motion of 0.01m or a torso rotation of 3°) related to the world due to the building quality.

The built-in cameras have the property that they are very sensitive to light. If the illumination of the environment is to low or there are shadows and noise on the top board side, it is challenging to distinguish different kinds of colored tokens. Furthermore it is not possible to extract depth information from a single camera, which requires special solutions while interacting in a three-dimensional environment.

Due to the the low point of view of the robot, tokens on the other side of the board are difficult to assign to a specific strategic position. Even for a human the classification of such an image is not an easy task. A misclassification could occur by occlusion of other tokens or by the robot itself or by perspective distortion. Perspective distortion leads to tokens that are visible in more than one mask and have a greater extent than expected. This results in an inaccurate estimation of the real token positions, which is required to grasp tokens. Furthermore the camera yields distortions that have to be corrected before the classification step.

The tokens were created especially for the robot. This led to game token cubes of 25 mm side length. However the height of the cubes evoke another issue in advanced game situations. For instance, if the inner square of the game board is occupied by tokens and the robot changes the table side, it tries to find the marker to relocate itself when it is approaching the table. The tokens on the inner square however occlude the marker from the robot's current position and view angle, and the marker cannot be found. Changing board, token or marker size just leads to new challenges.

Due to a small aperture angle of the camera and the close distance of the robot to the board, the token cannot be seen completely. Every strategic position has to be updated and sometimes it is possible to update more than one position at a time. So a special trajectory planner is required that reduces the steps of moving the head around to a minimum. The efficient planning is also required for token manipulation actions which should be also achieved by minimum effort of the robot.

Summarized challenges concerning the framework in the real world:

- self-Localization
- inaccuracy of the odometry and motion
- monocular vision
- camera calibration
- best size of marker, game board and tokens
- token segmentation on low illumination
- token classification
- token position estimation to grasp a token
- occlusion of tokens by other tokens and the robot itself
- grabbing tokens with three clumsy fingers
- efficient trajectory planning of interesting positions
- reduced stability on constant load - motors are overheating and lose stiffness, if joints are not relaxed
- limited network connection
- robust, simple, slim and modularized framework

All these challenges are discussed in Chapter 4.

Chapter 2

Sensors, Techniques, Hardware and Environment

2.1 The Robot

The humanoid robot Nao was developed by Aldebaran Robotics[2], a French company in Paris. The currently used robot platform is the Nao v3 RoboCup^{TM} version with two cameras mounted in the head module. The robot itself is equipped with many sensors and activators to perceive and interact with its environment. Additionally this retired soccer robot's hands have been upgraded to play board games. An overview of all its features can be seen in Figure 2.1.

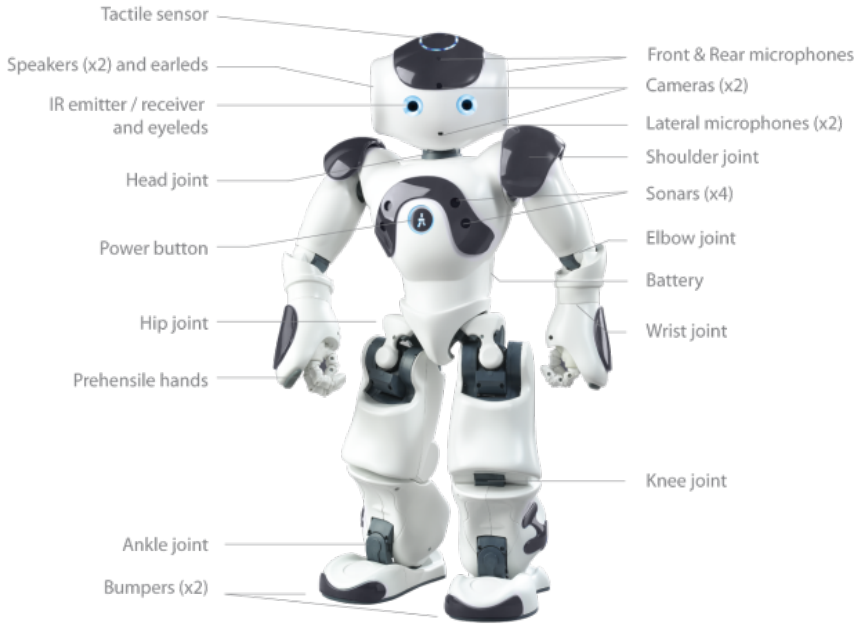


Figure 2.1: Overview of Nao's features from Nao Documentation 1.12.5 [4]

Focused on our project the most important robot specifications are listed below:

- height: 57.3 cm
- weight: 4.8 kg
- CPU: x86 AMD Geode LX-800 operating at 500 Mhz, situated in the head module
- 256 MB SDRAM
- 2 GB flash memory
- degrees of freedom:
  - Head: 2
  - Arms: 5 in each arm
  - Hip: 1
  - Legs: 5 in each leg
  - Hands: 1 in each hand
- 1 speaker in each ear
- 2 microphones
- network access by Ethernet or Wi-Fi (IEEE 802.11g)
- 4 force sensitive resistors on the bottom of each foot
- inertial unit consisting of a 2 way gyrometer and a 3 way accelerometer in the torso
- 1 bumper on the front of each foot
- 2 sonar emitters and 2 sonar receivers in a cone of 60° on the front chest
- 3 capacitive touch sensors on the top of the head module
- 8 RGB LEDs in each eye, 10 Blue LEDs in each ear, 1 RGB LED in the chest, 1 RGB LED in each foot
- operating system: Embedded Linux 32 bit with updated NaoQi version 1.12.3

All sensors, but the sonar, were tested for usability. Although the sonar would have been interesting for obstacle detection in the robots path it has not been used to the present day, because we assume that due to the thin board the sonar is too inaccurate for a reliable measurement. We also tested the microphones to use them for voice recognition, but the already implemented one does not work reliable and therefore was not used in the project.

The main work of this project is focused on the head module of the robot. The head contains two RGB cameras with a maximum resolution of 640x480 pixels or 30 frames per second. The 30 fps are only achieved on a lower resolution and only if the image is requested local and in its native camera color space YUV422. Both cameras are mounted in the front side of the head module. The bottom camera is located in the area, where a mouth of a human face would be, while the top camera is located in the forehead as visible in Figure 2.2.

Although there are two cameras in the head module, they are not usable for sterovision due to their non overlapping assembly. No points are visible with both cameras at the same time without a head movement, which would be required for stereo vision. Additionally the simultaneous use of both cameras is not possible.

The exact assembly of both cameras can be seen in Figure 2.3. The two identical cameras have a vertical field of view of 34.8° and a diagonal field of view of 58°. In the initial position of the robot, the bottom camera position in relation to the torso of the robot is 48.8 mm in front of the torso and 150.31 mm above the torso origin with a side offset of 0 mm. It is orientated with 40° from the horizontal of the camera

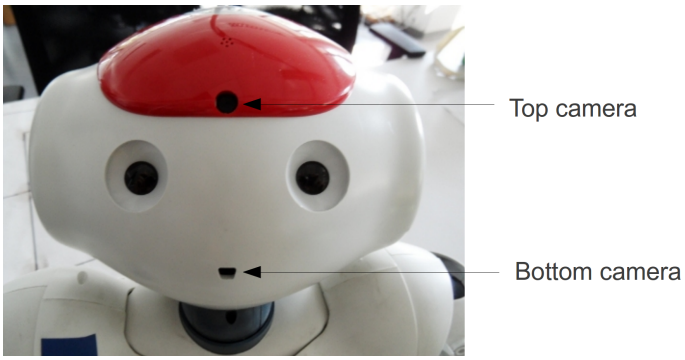


Figure 2.2: Camera module showing the two cameras

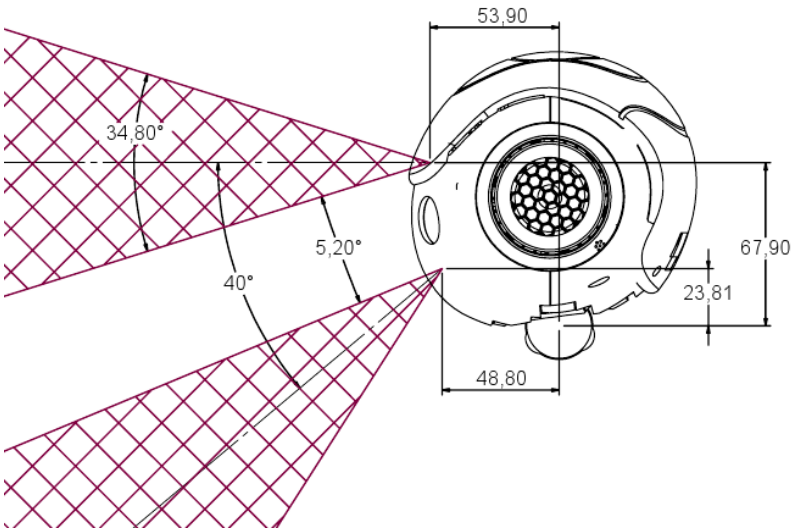


Figure 2.3: Head of Nao with the position and orientation of both cameras [3]. In this project the vertical field of view and the inclination of the bottom camera is of major importance, as they are often used for calculations.

origin to the ground. The roll (angle of x-axis) and yaw (angle of z-axis) is 0° at start and changes on head motions, while the pitch (angle of y-axis) is 0° and can never be changed.

There are three capacitive touch buttons at the top of the head, as shown in Figure 2.4. In our project these buttons were used to provide a simple user interface, for example the first time the front button is touched, the robots starts the game. The next time one of the buttons is touched, the game mode may be selected. The strategy is to keep the user input simple. The robot itself has no screen where the current state could be shown, but there is a text to speech proxy which comments its actions and tells the user the current functionality of the buttons. Furthermore the robot possesses 31 LEDs all over the body with 8 LEDs in each eye, able to emit red, green and blue color. The LEDs in the eyes are used to show that Nao sees the marker on the game board and each ear glows, if a player color has been activated.

Figure 2.4: Touch sensors on the head module.

Nao has two different network communication devices, that offer a wired and wireless connection. After the robot has been connected with a RJ45 cable to the network with a DHCP server, it receives an IP. The IP is required to connect to the robot and to use the NaoQi. The IP may be retrieved by one of the following options:

- pinging Nao's name. In our case this would be "amy.local" (default is nao.local)
- looking in the table of the DHCP server
- pressing the chest button
- opening Aldebaran's "Choreograph" and obtaining the IP from the connection manager. This is possible, because the robot registers when it is connected to the network

Some clients also support DNS which enables a connection with the robot name.

The wireless connection can be enabled by the web interface. The web interface requires an already established wired connection and is reachable by its IP or name. In the category "network" the desired network can be chosen. Although the wireless connection is comfortable, because no cables are required, the latency is remarkably higher and the bandwidth is lower than with a wired connection.

2.2 NaoQi, Python, C++ and the game engine

Nao has its own library, called NaoQi that offers a simple interface for higher commands in C++, Python and Urbi. The motion module of the NaoQi provides high level functions for walking. These basic walking procedures include

- walking forward and backward,
- rotating,
- strafing left and right, and
- walking simple curves.

At the beginning we tried to use C++ in ROS and the NaoQi, but as both are using different versions of the boost library, it is not possible to use them together. It would be possible to use a version, where both use the same boost library, but then one is restricted to a specific version. This was the reason why we switched to Python for the interface with the robot. Hence Python does not use a boost library, we separated the boost version of the NaoQi from our framework. Although Python offers the same interface functions for the robot and sometimes even an easier access, it is sometimes slower than C++, depending on the implementation. At first we though that the video stream transmission of the robot is bottlenecked

by Python, but soon we realized that the bottleneck is based on the color space conversion on the robot and the network connection.
The game engine framework also requires a specific boost library, which resulted in conflicts with our used ROS boost version. Since Ubuntu 11.10 the boost library version is 1.4.6 or higher, which is required by the game engine framework and therefore both frameworks are working together. We used Ubuntu, because of the good compatibility to ROS.

2.3 The Game Board

The white Nine Men's Morris game board consists of three unfilled black squares connected with a line in the middle of each side as may be seen in Figure 1.2.
At each intersection of two lines there is an ideal position. That results in eight ideal positions on each of the three squares, 24 in total. The inner square sides have a minimal distance of 58.5 mm to the center. This distance increases by additional 50 mm per square. Combined with the white border the square board has a side length of 376 mm. The height of the upper game board surface from the ground is 227 mm. This height has been chosen, because this way the robot is able sit next to the table with its feet below it. The upper game board has a thickness of about 16 mm and is held by four cuboid pillars with a side length of 33 mm. On the 16 mm sides of the top element some colored duct tape was mounted. This colored tape is used to help the robot in the relocation process, if the marker is out of range or not visible.

2.4 The Marker

In the center of the game board is a square AR marker with 70 mm side length, which is shown in Figure 1.1. This marker may be configured and tracked with the AR ToolKit [5], which is available freely, for non-commercial use under the General Public License. The thickness of the black border has to be 20 mm, independent of the complete size of the marker. The shape of the inner pattern directly influences the recognition rate. Easy patterns with thick and easy structures improve the recognition rate. The estimated detection distance of an easy pattern directly scales with the size of the marker, which can be seen in Table 2.1.
Recently ROS became very popular, so there is already a wrapper that provides a few functions of the AR ToolKit for ROS. This wrapper is called "ar_pose" [16].

Pattern Size (inches)	Usable Range (inches)
2.75	16
3.50	25
4.25	34
7.37	50

Table 2.1: Tracking range for different sized patterns [5]

2.5 Game Tokens

Although the robot is able to grasp objects, there are restrictions due to its three fingers. In our case, the common flat Nine Men's Morris tokens of black and white color are replaced by colored modelling clay. These colored tokens have the shape of cubes with the edge length of 25 mm. This size has been chosen in order to achieve the best trade-off for grasping in relation to occlusion of the marker or other tokens. Two of those tokens of different colors can be seen in Figure 2.5. The purple color is not used any more, because the blue stripe on the border of the board is too similar. For detailed information about the cubes and colors, see Section 5.8.

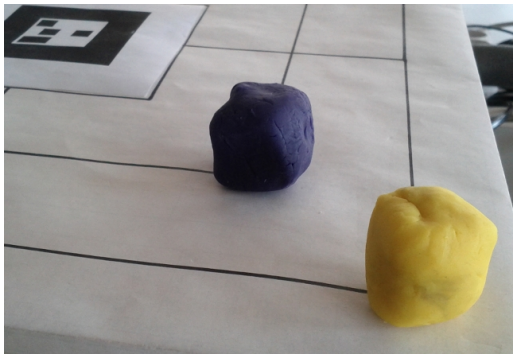


Figure 2.5: Two sample tokens of yellow and purple color. The yellow color has a unique range in the HSV color space that is not occupied by any other objects on the game board. Ahead from the other colors, the yellow color is detected best.

2.6 Hardware - Processing Notebook

The created ROS framework is not running on the robot itself. All the necessary computations are done on a provided notebook. The sensor and activator information are then sent via LAN or WLAN. The notebook itself is powered by an "Intel Core 2 Duo" P8600 with 2.40 GHz and the main memory modules provide 3.9 GB of fast flash memory. Due to certain 32 bit libraries for the robot, ROS and the AR ToolKit a 32 bit Ubuntu 11.10 was used, which is best compatible to both frameworks. The compiled project itself has a size of approximately 150 MB (5 MB source only) and could also be copied to the robot which has a flash memory module of 2GB. The complete gaming database has a estimated size of about 100 GB.

Chapter 3

Related Work - Research

The chapter deals with other work in related domains that inspired and influenced specific methods that were developed or adapted for this framework. It is focused on other game playing robots, solutions for localization with single cameras, artificial gaming intelligence, vision-based grasping and possibilities to segment camera images or detect colors.

Localization using a single camera
Liemhetcharat et al. [30] worked on a localization method with the humanoid robot Nao, only using the internal camera and the odometry. Due to a limited field of view, the robot is not able to see all objects continuously. In order to localize, different observations in the world are made over time. The used world model provides objects, even though they are currently not visible. A Monte-Carlo localization as proposed in [40], to estimate the robot's pose in the world, has been used. We decided to use an AR marker, which offers an absolute position that is fused with an odometry update. Additionally we used the camera to estimate torso rotations, which are also fused with the robot's pose.

During movements, Nao's camera images are blurred, depending on its motion speed. Hornung et al. [24] proposed an approach of learning the influence of motion blur to improve localization. The robot learns a trade off between velocity and localization accuracy. A very exact localization to grasp tokens is required, because an offset of a few millimeters leads to a fail grasp. Hence we decided to use only camera images, while the robot is steady and an AR marker to deal with accumulating odometry errors.

The work of Davison [14] deals with a robust localization method, only using a single camera. Furthermore the presented approach is running in real time with very sparse prior knowledge by a hand-waved camera on a common desktop computer. Automatic reconstruction from motion of camera trajectories in unknown scenes was already presented in [18], where offline algorithms were used to compute this task. Online algorithms are far more challenging due to their short minimal available processing time. Davison [14] used a camera with 30 frames per second, which implies a maximum available processing time of 33 ms. The framework is initialized in a resting position with a known object (A4 piece of paper). Afterwards the camera can move smoothly but rapidly and freely in a three-dimensional space. The goal is to continuously estimate the camera's position. This means that repeatability is very important, because the framework should yield the same result after 10 seconds, as it does after 10 minutes. The deviation of an estimated position to a position, which has not been seen for some time, due to camera motion, is called motion drift. Real-time self-localization and mapping (SLAM) has a problem of motion drift which has to be modeled and handled. Davison [14] combined an extended Kalman filter with various approaches of [17] and [29] to reduce computational complexity to handle the propagation of first-order uncertainties. The uncertainties itself were represented with a covariance matrix. The update of a full covariance matrix takes time, which is a limited to 33 ms. The previous work of Chiuso et al. [12] already presented a real-time, full-covariance Kalman filter-based approach to structure from motion. This approach only allows slow camera motions and no re-acquiring of older frames, which leads to motion drift.

Davison [14] added a motion model to their approach modeling a Gaussian profile. Assuming that accelerations occur, but very large accelerations are unlikely, gives the model additional robustness. They have modeled the velocity of the camera, while other structure from motion models start again from scratch with each image.
To avoid motion drift, long term landmarks are required. These landmarks are recognized again after a long period of neglect. In [15] an approach for large image patches with strong salient features that could serve as long term landmarks has been proposed. Three-dimensional depth information cannot be obtained by one image, therefore the initialization of features is a difficult task. Consequently only a three-dimensional ray is initialized that maps from the camera center through the image pixel. The feature point has to lie on that ray. On this ray assumptions are made that resemble a one-dimensional particle distribution in a range of 0.5 m to 5 m. After the camera has moved, all particles are projected back into the image, which offers an elliptical search region for each particle. This results in a new probability estimation of all particles. The new distribution forms a peak and represents a specific distance. Speed improvement is achieved by eliminating the weakest particles.
A map that is created by these particles has to be managed properly, thus only reliable features shall be kept. The most interesting features in a randomly chosen image window are detected with the interest operator of Shi and Tomasi [37]. Bad or often occluded features have to be eliminated. A feature is deleted, if it is not detected, while it should be visible. Often, appearances of similar clutter and landmarks lead to complete failure on mismatches.
The computation of the framework was done on a 2.2 GHz Pentium processor and required approximately 25 of 33 available ms to process an image.
This approach inspired some designs and implementations of projections and feature detections, which were used for the classification of the game tokens and the visual odometry.

Another approach of estimating the camera pose in an unknown scene in real time was proposed by Klein and Murray [27]. In contrast to SLAM they presented a method called parallel tracking and mapping (PTAM). This approach is designed to track a hand-held camera pose in a small AR workspace. They split the tracking and mapping in two threads to take advantage of multi core processors. Whereas the tracking thread finds keypoints and updates the camera pose for every frame at 30 Hz, the map is only updated at keyframes, because the update of the map can take much longer than 33ms. To track keypoints and estimate the camera pose, they use a zero-mean sum of squared differences (SSD) score to compare gray levels of image patches. A three-dimensional reconstruction of a scene requires multiple observations from different camera poses that demands camera translation. Due to this translation of a camera, a virtual baseline is created that can be used for reconstruction. There are several reasons why this approach is not well-suited for our project. First of all, the motion blur, which the images of the robot suffer during translation. This makes the tracking of keypoints very unstable. Secondly, the approach was developed to view a small workspace from the outside, but in our case the robot is placed inside the workspace and most of the features lie on the outside. Due to the limited robustness against motion blur the keypoints (and consequently the camera pose) cannot be tracked long enough to establish a baseline which is wide enough to obtain a new keyframe.

Segmentation and color classification
Dahm et al. [13] discussed different color spaces for robot soccer. A chrominance color space transformation to facilitate the segmentation step was proposed. An adapted chrominance space as in [39] was used, where a “blue channel” was added. The original space that has been used for face recognition, did not comprise blue color values. The idea of the approach is to split the chrominance space in subspaces, where each subspace represents exactly one color. This facilitates the detection of a color from a hyperplane to simple threshold values. The approach has not been further examined since distinctive color values of the game tokens in the HSV color space were chosen. The decision is further discussed in Section 5.8.

The work of Harahap et al. [20] deals variable illumination. Three different illumination normalization methods, as a pre-processing step in tasks of robot soccer, were compared. According to [20],(1) histogram equalization is widely used, because of its simplicity. Sometimes along with the histogram equalization, (2) gamma intensity correction and (3) logarithmic transformations are used. Inspired by the idea of a consistent illuminated image, histogram equalization was implemented and tested in the image preprocessing

step. We expected a result, which was more invariant to illumination changes, but the normalization just increased the contrast level. Therefore the normalization step was revoked (see Section 5.6.3).

A segmentation procedure usually contains thresholds somewhere to divide different colors. The work of Bruce et al. [11] compares different color spaces and introduces an approach for a fast color segmentation. A specific color in the RGB color space with variations in the brightness is represented by a conical body and cannot be modeled with simple thresholds. However, the HSV and YUV color spaces encode the chrominance in two channels and the illumination in one channel. So a specific color can be found within a threshold for the chrominance and nearly the complete illumination channel.
Common color thresholding, also considering illumination would require six thresholds that have to be compared with the current pixel values, which lead to six comparisons per color and pixel. Bruce et al. [11] propose an approach to segmented up to 32 colors by three binary ANDs only . Considering the following example for a YUV color image, discretized to ten color levels, the orange color would be represented as:

$$YClass[\,] = \{0,1,1,1,1,1,1,1,1,1\} \tag{3.1}$$
$$UClass[\,] = \{0,0,0,0,0,0,0,1,1,1\} \tag{3.2}$$
$$VClass[\,] = \{0,0,0,0,0,0,0,1,1,1\} \tag{3.3}$$

If a pixel with color values (1,8,9) is a member of the color class ”orange”, the expression of YClass[1] AND UClass[8] AND VClass[9] is evaluated, which resolves to *1, true*. Considering blue as a second color, both colors can be evaluated with the same procedure.

$$YClass[\,] = \{0,1,1,1,1,1,1,1,1,1\} \tag{3.4}$$
$$UClass[\,] = \{1,1,1,0,0,0,0,0,0,0\} \tag{3.5}$$
$$VClass[\,] = \{0,0,0,1,1,1,0,0,0,0\} \tag{3.6}$$

Both arrays can be combined into one, where each array element consists of a 32bit integer.

$$YClass[\,] = \{00,11,11,11,11,11,11,11,11,11\} \tag{3.7}$$
$$UClass[\,] = \{01,01,01,00,00,00,00,10,10,10\} \tag{3.8}$$
$$VClass[\,] = \{00,00,00,01,01,01,00,10,10,10\} \tag{3.9}$$

The first bit now represents ”orange” and the second bit ”blue”. If again the same color values (1,8,9) are evaluated, the result is *10. 10* indicates the color ”orange” but not ”blue”. ”Blue” would be *1* on the second bit, like *01*.
Subsumed, if the image is stored in such an array structure, up to 32 colors can be segmented easily with three binary ANDs in contrast to a standard procedure with six comparisons for each color. This is a huge difference in computation time, if more colors have to be segmented.
The approach of [11] was not used directly for the segmentation, because in our project only three colors are tracked and therefore the speed increase to simple thresholds would be minimal. Additionally we would need to convert our image structure to the one presented in this approach to make use of it. This conversion also consumes processing time. However the classification method that compares images by binary ANDs was influenced by this approach and is described in Section 5.6.6.

Vision-based grasping
The work of Höll and Pinz [22] deals with autonomous grasping of objects from a table. Hand-eye coordination of the robot was focused, while many other challenges influenced their work. These influencing factors include:

- limited processing power
  All computations are executed on the 500 MHz processor of the robot. In contrast to our approach they do not require a large database that would not fit on the memory module of the robot. The processing power does not affect us, because we compute all tasks on a remote machine. However, we need to transfer the images and joint values to our remote machine, which require a continuous network connection.

- inaccuracies in repositioning
  Due to the limited production accuracy, executing small translations and rotation with the robot is nearly impossible. Additionally, larger rotation estimations are imprecise, depending on the underground.
- instability on longer operations
  After continuously operation, the joints of the robot are overheating and the NaoQi automatically reduces the stiffness to avoid motor damage. If the robot moves in that state, it may break down or fall on the ground.
- self-occlusion
  For a visual-servoing process it is essential to see the target and the hand. As soon as the robot is next to the object that should be grasped, its own hand occludes the target most of the time. In our project visual servoing was not used, because a free sight to the token and hand cannot be guaranteed. Instead a failed grasp is being repaired.
- limited grasp-ability
  The robot has three fingers that are controlled by one wire-rope. There are no touch sensors to know when something is between the fingers.
- monocular vision
  Stereo-vision is not possible, as both cameras cannot be used at the same time. Additionally no overlapping region is available, which is required for matching the images to obtain stereo vision. Therefore special solutions to interact in a three-dimensional space are demanded.

The board plane is estimated by projecting a plane in the hand of the robot that is parallel aligned. Additionally the ground plane is calculated the same way it is done in our project, by computing the transformation from one of the feet. The feet of the steady Nao are assumed to be parallel to the ground.
Höll and Pinz [22] use the transformations provided by the NaoQi and the knowledge about the orientation to the table to project the finger tip of the robot's thumb onto the table plane. Nao's thumb is essential for a successful grasp. The projected thumb is moved to a target position that lies in the center of the bottom line with a little offset to the top side of the blob, depending on the arm position.
Until the projected thumb reaches the target destination, a visual servo control is used to move the arm. Höll and Pinz [22] achieved good results with their autonomous grasp approach for all the challenges they faced. They achieved a successful grasp rate of about 74 %, which achieves a better score than our approach.
In contrast to Höll and Pinz [22], our framework has a complete kinematics that plans a path to the target destination. We believe our solution that only takes a few seconds for a grasp, is much faster than using a visual servo control that has to wait for feedbacks from the vision. This was also an important factor for us to improve the game experience, because the robot has to grasp quite often and the human player might get bored while waiting.

Game intelligence
There has been an approach to play the board game Pylos with an autonomous robot by Aichholzer et al. [1]. Pylos is similar to Nine Men's Morris, as it is also a game of full information for two players, which means that all data is available for both players. A Katana robotics arm with six degrees of freedom was used, while the recognition used a standard firewire camera that provided color images. The game strategy for Pylos has been solved in order to have the optimal move for every game state, which is saved in a 30 GB database. Impossible moves of the opponent are detected by a crosscheck with the possible successor states that are saved in the database. In our approach we used the same game engine framework, but with a database for Nine Men's Morris. This enables the robot to always make the optimal move.

Other related humanoid gaming robots
The team Sun et al. [38] managed to develop a humanoid life-size robot with a height of 160 cm that is able to play ping-pong versus a human opponent. To achieve this, a robot, which is robust and flexible enough for walking and playing table tennis was needed. Moreover fast control algorithms and real-time control with efficient planning are required that also watch the stability of the robot, which is very challenging on high-speed motions of the robot arm. In addition, a very accurate perception, which identifies the moving

ball, predicts its motion and localizes the robot is required. In contrast to our robot, the vision consists of two on-board cameras, but due to vibrations and a short baseline of 10 cm, it is not accurate enough. Thus two additional cameras with a baseline of 110 cm are mounted above the ping-pong table. Whereas the overhead cameras are responsible for the ball detection, the onboard cameras are in charge of the real-time self-localization. The data of the overhead cameras are processed by an external computer and sent wireless to the robot. We could not adapt anything of this approach, because their used hardware is totally different.

Tomlinson [41] from the Machine Intelligence and Pattern Analysis Laboratory at the Griffith University researched on human robot interaction for board games. A framework for a Nao has been developed with an artificial intelligence for a domino player. Visual recognition of the domino tokens and the computation of the next move has been focused. The setup consists of a frame that holds the domino tokens in front of Nao's camera. The robot uses its arm and speech to indicate which token should be used next, but is does not walk or grasps any tokens. A human has to execute the indicated move. In our approach we want to be as autonomous as possible and we did not find any published work to this topic. Therefore this research did not contribute to our work.

Recently we got information about a company [25] that developed a connect-4 playing Nao. There is a short demonstration video on their website that shows Nao making a move. A big blue connect-4 game frame in front of a white background is used for the setup. The robot is initialized in a distance from where the whole game frame can be seen. Due to the fact that the frame size is known and completely visible, the distance to that board can be calculated for simple localization. In the robot's turn, it approaches the frame, opens its hand and waits for the human to hand over a token. Then it moves its hand above the frame slot, tracking the hand position with the camera. Nao adjusts the hand position by visual servoing until it is sure that the token fits in the slot. Afterwards it moves back to see the whole board. We assume that a board rectification for an abstract representation is applied. The fact that the complete game state can be checked in one image, while seating with relaxed joints, enables the robot to know when the opponent made a move.
In our project we have to grasp some token positions, where it is not possible to see the token to adjust the hand position. In contrast to the connect-4 playing Nao, we need to grasp tokens. A successful grasp requires a very precise positioning of the thumb, which is essential for the grasp. Due to self-occlusion the important position is not visible. Additionally we have the a horizontal game field instead of a vertical one. The whole game board cannot be seen in one image and it is far more distorted due to the perspective. Furthermore Nao has to be upright to acquire the current game state or execute any other action. The time slot when the human player has to make a move, is the only time to relax the robot. The relax is essential for continuous operations, otherwise the motors are overheating and the robot loses stability. This is the reason why we do not have an automatic recognition for the opponent move.

Influence of the robots appearance and behavior to humans
Due to the fact that Nao looks like a person, there has been a research to investigate, which conditions are responsible to make it appear human. In [33] the influence of the appearance and behavior of robots, using a machine-like robot and the Nao as shown in Figure 3.1, has been researched. The behavior of the robot has a significant effect on the ability to deliver information. There has been a tests on 40 students that were divided into four groups, where the robot tells something in two difference voices, in a text to speech and a human recorded voice. The results of these tests stated that it is easier to listen to the human voice to deliver information, independent of the context. Additional important factors to make a robot look like a human are smooth and natural motions, speech recognition to support communication, a face where humans can look at and a humanoid shaped body. Based on the reasearch of [33], we used Nao to enhance the game experience of the human player to the robot. Thus a humanoid robot was used, instead of a robotics arm, which does not trigger the same emotions of the human player.

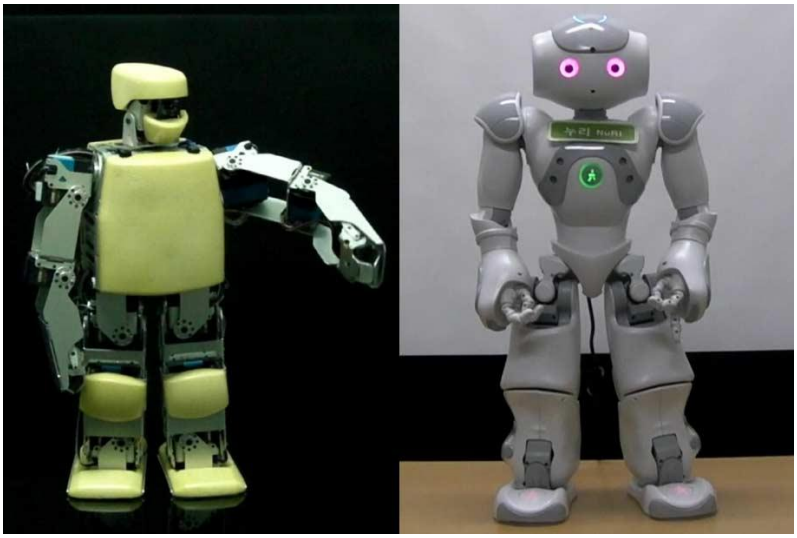


Figure 3.1: Machine-like and human-like robot [33]

Chapter 4

Concept to solve the Task

This chapter discusses the concept of our developed framework to solve all challenges. The solution to the challenges is described in detail in Chapter 5.

4.1 Overview

The basic idea is to play a board game with a humanoid robot. At the beginning of the project two questions had to be asked:

- *"What do we want to achieve?"*
- *"Which resources are available?"*

From the early beginning on it was clear that we want to develop a framework for an autonomous humanoid robot that is able to play a board game against a human. The human should think of the opponent more as an entity than as an object, because while playing games, humans feel many emotions. For them, it is more easy to gain an emotional state of playfulness, when they do not believe to play against a machine. The influence of appearances and behaviors of machines have been researched in [33]. It should be possible to adjust the difficulty to always have a competitive opponent. Furthermore it should be simple to expand the framework that two robots can play against each other. This way, a new section at the Robocup [35] could be created, where robots play against each other in peaceful board games. In future work it should be possible to play a wider collection of games in addition to Nine Men's Morris with this robot.

Concerning our resources we were limited by the only available humanoid robot, which fulfills all requirements - Nao. This robot has a large repertoire of sensors, activators and degrees of freedom. Due to the hand upgrade it is possible to grasp something with the robot. It is able to walk around, move its arms and head. The built-in color cameras would serve for localization and recognition for its environment.

4.2 Basic Nine Men's Morris Rules

Nine Men's Morris is a strategy game for two players that has very simple rules. Several items are required to play. This includes a game board as already shown in Figure 1.2 and nine game tokens of the same color for each player, as shown in Figure 4.1. The game board's 24 ideal locations, which are defined by intersections, represent the valid positions for tokens in the game.

To win the game, the players have to create special structures to take away the opponent's tokens. Every player has to take away seven of the opponent's tokens or trap them to win the game. The tokens are

Figure 4.1: Final game tokens for both players

considered to be trapped, if no valid move is possible for any token on the start of that player's turn. It is a game of full information, because there are no hidden cards or any dices that would randomly influence the game.

The players start by taking turns in placing all nine tokens on the board on empty positions. Every time a player achieves to get three own tokens on one black line, a so called mill has been created. As the result of creating a mill, the current player is qualified to remove a token of the opponent player on the board, for the rest of the game. Tokens in a mill are save from opponent attacks.

If all tokens are placed the second game phase starts. The same player that started placing the first token, starts moving a token. A valid move is on a line from an intersection to the next unoccupied intersection. The strategy is to move the tokens in a way to create a mill to take away opponent tokens. If a player is unable to make a move, the opponent player wins the game.

The third game phase starts, when one player has only three tokens left. Every player who has only three tokens, can jump with a token to a free position everywhere on the board. By jumping, one can interfere in the opponent's plans or easily create a mill. Sometimes it happens that both players have three tokens. In this case draw games are common, if both play perfectly.

4.3 Development Environment

The concept was implemented in C++ and Python, because these two are supported by ROS. Even if ROS is not directly required for the robot, it offers tools that makes the development much easier.
Different boost versions of the NaoQi and ROS C++, are not compatible if used simultaneously. There is no simple method to get them running with different boost libraries. Thus for the direct robot communication only Python was used. By using Python, which does not uses a boost library, the different libraries run in strictly separated files. This ensures the functionality of the framework, if one module is updated or uses another boost library version. Each framework can use their own boost version. However, since the game engine framework was included, which also requires a boost library, at least the boost version 1.46 is required for our framework.

The architecture of ROS consists of

- **stacks,**
- **packages, and**
- **nodes.**

Each stack can be seen as a module, which fulfills a higher level task, for example "connect to a camera and visualize images". Additionally each stack consists of packages that split the task in smaller subtasks. Furthermore each package consists of nodes. These node split the complexity of the task again and each node can handle one specific or a various number of simple tasks.

ROS offers easy communication strategies between nodes. Therefore a modularization is simple and individual nodes may be replaced without compromising the remaining framework. There are three main communication possibilities between nodes:

- **Publisher-Subscriber**: One node publishes information with a topic name. Any node that wants to have the information subscribes to the topic name. The publisher needs to be created some time (about 500 ms) before the first message is send to be registered in the ROS system, otherwise the message will not be sent.
- **Services:** A service requires one node that has the function of a server and offers the service with a service name. Another node (client) may call the service with some individual data and get a response to its special request. This resembles a double sided handshake procedure in contrast to the publisher-subscriber principle.
- **Actions:** Actions are similar to services, but usually they take longer. This is a possible option for robot movements, where anything else may be computed in the meantime. Actions support the possibility to request the current completion state of the given task. In our project no actions were used. It was always necessary to wait for movements to be finished until further data can be processed.

Conclusively ROS offers an easy way to transform between different coordinate systems. This is essential or at least very helpful for robot systems. If the robot has to grasp a token on the other board side, it has to walk. While the token position, related to the board, stays the same the position changes in relation to the robot.
A working transformation stream consists of broadcasters that update all required transformations. Each broadcast command offers a transformation between two systems. All broadcasted transformations are combined in a stream. The systems are arranged in a tree shaped structure. This structure implicitly includes that there are no circles allowed in the transformation stream. Otherwise all transformations in the tree are not unique anymore and the transformation will fail. To use the transformation a listener is required to transform a point or pose. The listener is subscribed to two frames and saves the stream for some seconds. If it is required, it is also possible to use an older transformation. Transformations are described in detail in Section 5.3.4.

At the start we used an already existing ROS-stack for Nao [23] that is maintained by Hornung. This stack contains a script to connect to the robot, a joystick controller to walk and move the head and a simple URDF model. This was very useful for our first steps. In the continued development process, we used less and less of it. At the end only the connection to the robot and the robot model are being used.

4.4 Basic Concept

For this project is was necessary to develop a framework that should cope with a large variety of challenges, the robot faces, if it plays this game. I focused on vision tasks, localization tasks and the intelligence behind the robot's actions and the framework itself. The kinematics, walk and grasp planning was solved by Klöbl [28]. The used game engine itself, represents the robots game intelligence was developed and provided by

another group. The game engine consists of two parts. There is a framework that serves as an interface to handle a variety of different board games and the game database itself. The database holds every state of a Nine Men's Morris game, so for each state the next best k-solutions can be requested. The game engine framework was developed by Regenfelder [34], whereas the database of the game moves was developed by Staber.

Many challenges have to be considered in the framework. There were different strategies developed in order to solve them.

- **self-Localization**
  The robot has to use the camera and a marker that is mounted on the game board to establish a connection between its joints and the environment. The internal odometry is the source for all robot joints.
- **inaccuracy of the odometry and motion**
  The inaccuracies have to be compensated by a control loop that uses the camera image to correct movements.
- **monocular vision**
  Additional information of the environment, like the board height, has to be used to acquire three-dimensional positions with a single camera. The trigonometric relation has to be derivated.
- **camera calibration**
  A calibration target has to be manufactured to receive the camera parameters, using a calibration toolbox or method.
- **best size of marker, game board and tokens**
  Empirical tests have to be performed, to retrieve the best size for all environmental objects.
- **token segmentation and illumination**
  Tests concerning the illumination and the best token color have to be made.
- **token classification**
  A robust approach for different poses of the robot to classify tokens to a strategic position has to be found.
- **token position estimation, to grasp a token**
  A reliable three-dimensional position in the game board system has to be estimated, using the camera pose.
- **occlusion of tokens by other tokens and the robot itself**
  Occlusion during a grasp is not preventable, therefore the previously estimated position has to be precise and a confirmation step and repair on a fail grasp are required.
- **grasping tokens with three clumsy fingers**
  This is an advanced grasping task that we have to deal with, but was not focused in this work. The most reliable position for a token has to be found to increase stability.
- **efficient trajectory planning of interesting positions**
  The recognition of the game state requires multiple images. Between two images a movement is executed, which has to be considered in the new image. A strategy is required to obtain the correct new game state with as little images and movements as possible.
- **reduced stability on constant load**
  Nao's motors are overheating and they lose stiffness, if the joints are tense for a few minutes. The reduced stiffness is a problem of the robot itself. The resting time of the robot between actions could be increased to spare the motors.
- **limited network connection**
  The network traffic has to be reduced to a minimum. All nodes, which access the robot or produce traffic have to examined.

4.5 Framework architecture and Features

The stack of our framework is divided into seven packages that handle all required tasks for this project. The relation between the packages can be seen in Figure 4.2 and are described briefly in the following.

- The **high level control** uses behaviors, which handle database requests to offer high level functions only, for example: ”move token”.
- The **planners** contain mid-level behaviors and planners, which use vision and motion tasks, for example: ”calibrate color”.
- The **game database** comprises the game engine framework and database with a wrapper library that offers its core functions for ROS, for example: ”get next game state”.
- The **world model** is the knowledge base, which stores game, robot and environment data.
- The **helpers** are a set of useful functions and definitions that are frequently used by more than one package.
- The **motion** executes movements and checks sensor values, for example: ”set all angles”.
- The **vision** handles all tasks related to the camera, recognition and image analysis, for example: ”get camera image”.

Our stack uses several standard stacks of the ROS framework, which can be found in Section 5.9. If another package is required inside a package, a dependency with the name of the other package has to be inserted in the *manifest.xml* file of the package.

In order to achieve all goals, there will be given an overview of the functionality of each package. While the *vision* and *motion* are low level interfaces, the *world model*, *planners* and the *game database* are mid level interfaces, based on these low level interfaces. The package *high level control* contains high level functions based on the *planners* and *world model*. A model of the communication between the packages and the robot itself can be seen in Figure 4.2.

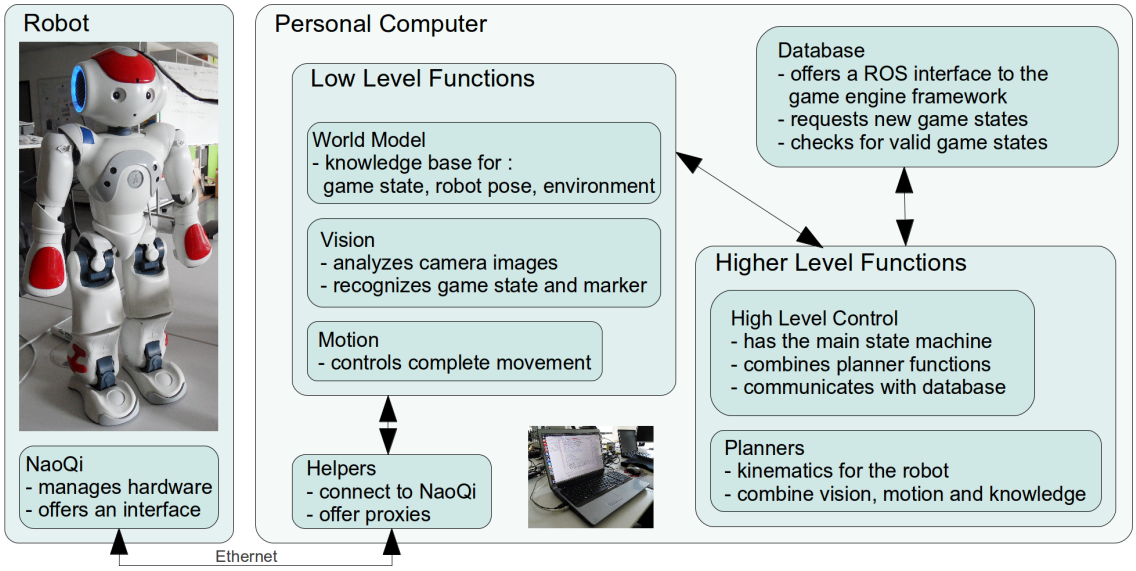


Figure 4.2: Model of the communication between the packages of the framework and the robot

While all nodes each accomplish a simple task, they need to communicate to complete a higher, respectively the complete task.

If two nodes want to communicate with a service, the client needs to be sure that the service is available before calling it. Otherwise the call may fail and the node returns an error or in the worst case, it terminates itself. A service always consists of a service server and at least one service client. Each service client may call the service to send a request and receive and answer. Every node that uses service clients is waiting at startup until the service is available. This greatly increases the robustness of the whole framework, because it does not matter which node is started first.

4.5.1 Package: high level control

The high level control offers the highest level functions of our framework combining behaviors and using planners. This package also contains the main state machine that handles the execution procedure of the robot. Additionally it sends requests to the game database to evaluate the next best step. The different high level states the robot has to achieve in order to solve the task can be split up into:

wait: The robot waits until the opponent is ready to play or finishes their turn.

init: The init state initializes many procedures. It turns on the motion, finds the marker, locates the robot in the world and initializes the player colors for the current light conditions.

get game state: Due to a low aperture angle and close distance to the board, it is not possible to receive an image of the whole game board. In this state the robot looks at different positions on the game board to combine all images to acquire one solution for the current game state. Each position is evaluated on whether or not it is empty, occupied by player one, player two or if the position is unknown.

edit token: This step uses behaviors, the kinematics, walk and grasp planner for a token manipulation. This is a challenging task for a robot with 25 degrees of freedom.

help: The help state is called, if something unexpected has happened and the robot needs human help to decide what should be done next.

These higher level states should cope with a large variety of situations, happening during the game. Still we assume to be under laboratory conditions, like the light should not change dramatically and the human should not interfere too much in the robots actions. After a game has been started, the robot is able to play a whole game with a human player with very restrictive support. This support is restricted to hand over or take tokens out of the robot's hand, because it does not know where to put them. A voice synthesizer is used by the robot to inform the human player about the robot's actions. For debugging, there is a large test class to test single components of the framework.

4.5.2 Package: planners

The planners offer an advanced interface for mid-level functions. They call and evaluate vision and motion tasks to provide combined functions. The planner package was developed in cooperation with Klöbl [28] and is able to handle the following tasks to

- handle the complete color calibration.
- analyze the game board and return the current game state.
- look at ideal positions.
- find the marker.
- place/take/move tokens to/from given marker positions.
- execute a walk to the given marker destination, avoiding the table.
- check or wait for tokens in the hand, also including the movement to the check position.
- relax the entire robot.
- include the kinematics class of [28], which computes the angles of joints, which are required to reach a specific position.

4.5.3 Package: game database

The game database offers a service to communicate with the Nine Men's Morris' framework and database. If a request is sent, it is converted into a specific string, required by the database. The database returns the best next state to the current given state. The database should also be able to return the k-best states, but this only exists in theory, because the computation of the database was not complete at the end of this project. The k-best moves could then be used to adjust the game difficulty. We believe, it would be frustrating for the human opponent, if the robot plays perfectly and wins every single game.

4.5.4 Package: world model

The world model also represents the knowledge base, including knowledge about the following to

- store the state of the motion, vision, camera, arm and all tactile sensors (bumpers, chest button and head touch sensors) and the game.
- store probability and occupation of each token on the game board.
- hold ideal position to grasp a specific token in all systems, as well as dimensions of the table and the ground.
- compute next interesting position to look at, as well as which arm to preferably use in this process.
- fuse all transformations and sensor sources.
- hold all joint values.

Additionally the world model has transformation scripts, which are of great importance. One of these scripts also handles the self-localization of the robot. If the marker is not visible, the robots location is updated with its internal odometry. The position is broadcasted as a transformation between the torso and the marker. This is the most important transformation for the localization to obtain the robot pose in relation to the marker. Every odometry produces inaccurate localization values. The robot's rotation around its vertical axis produces the most inaccurate value, because sometimes one of its legs slips on the ground. This localization error has to be corrected, otherwise the robot bumps into the table. Therefore we used the AR marker, which updates the robots absolute position, if it is visible. The robot can switch between two cameras. If the camera is switched, the transformation to the marker also switches to ensure that the pose of the torso is the same.

4.5.5 Package: helpers

The helper package is a package to simplify the framework usage and is intended to offer

- custom messages,
- services,
- error codes,
- task IDs,
- parameters,
- frequently used functions and
- an interface to the NaoQi for all packages of the stack.

4.5.6 Package: motion

The motion is classified as a low level package for communication with the actuators of the robot and has been developed and implemented by Klöbl [28]. It covers the following tasks to

- initialize the robot motion by moving all joints to an initial position.
- offer a lot of services to move all desired joints to a specific angle value.
- offer a relax function to lower the stiffness. This should be used, if the robot is idle for some time.
- offer a command line interface to control joints by reading or setting positions or angles for joints.

4.5.7 Package: vision

The focus of this thesis was especially on the solution of the vision tasks. The vision should be based on a clean and robust implementation, which is capable to handle a range of tasks and inconsistencies in images. The main challenges, that need to be solved in order to play Nine Men's Morris, are summarized in the following to

- subscribe to a video module to receive images.
- offer a service to assign a desired color to a player.
- detect a blob image of the previously initialized colors.
- allocate the detected blobs to strategic positions.
- estimate the real token position on the board, if a blob is found near an ideal position.
- publish the current board probabilities to the world model.
- offer a service to confirm a grasp or wait until a token is in the hand.
- offer a service to confirm a place or take.
- compute the horizontal offset of two images. This is used by the walk planner to compute the rotation error and is similar to a simple visual odometry.
- detect an AR marker in the camera image and compute its transform to the camera.
- offer a service to calculate the distance to the blue stripe of the table.

Chapter 5

Solution to the Task

As mentioned in the concept in Chapter 4, the framework consists of several packages to modularize the framework and solve the task. In this chapter each package and its features are discussed in detail. Due to dependencies between the nodes of the packages, based on service servers and publishers, each node waits until the other side is ready to send the required information. This is essential for a robust framework, because nodes with service clients fail, if they call an inexistent service. The framework is designed to be able to start nodes in any order, as it will synchronize them automatically. The overview of the dependencies is shown in Figure 5.1

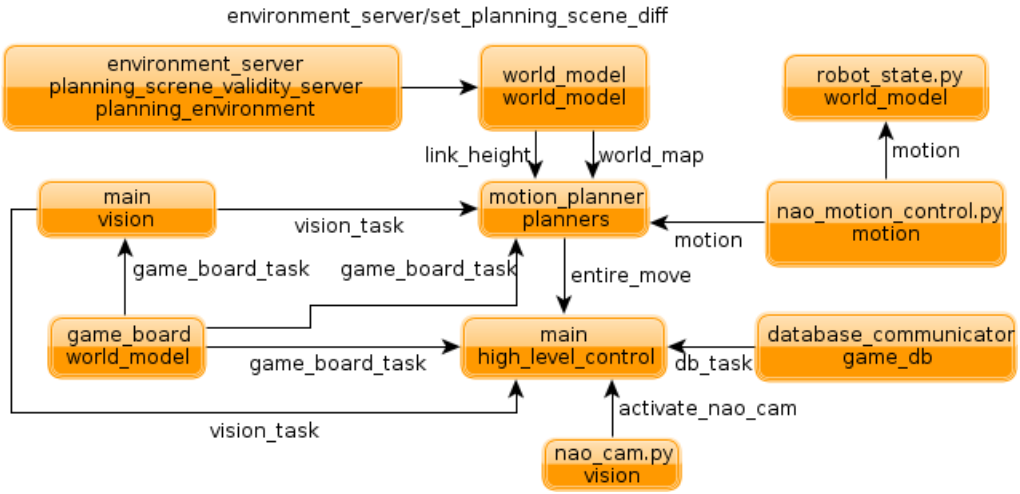


Figure 5.1: Dependencies of nodes, waiting for another node to offer their service server. The shaft of each arrow represents the server and the tip represents the client. The top description refers to the node and the bottom to the package name.

Although we took care of the dependencies between all required nodes, a“launch-file” has been created that is able to start all required nodes for the whole framework, except the main executable for the state machine (see Section 5.1) and the interface for the database (see Section 5.4. In the following sections, each package has figure with a summarized overview of its core functions. These figures are divided in smaller areas, which represent libraries, classes or nodes. Nodes include classes, libraries, provide services, subscribers and are executable.

5.1 High Level Control Package

The tasks of the package are shown in Figure 5.2. As the high level control is our highest abstraction layer it has the main executable, which contains the main state machine. On execution the main node waits until all other required nodes of the framework are started and executes the state machine to play a complete game of Nine Men's Morris. The states are visualized in a flow chart in Figure 5.3 and are explained below. If everything worked as expected the robot will terminate the program itself, when it has lost or won the game.

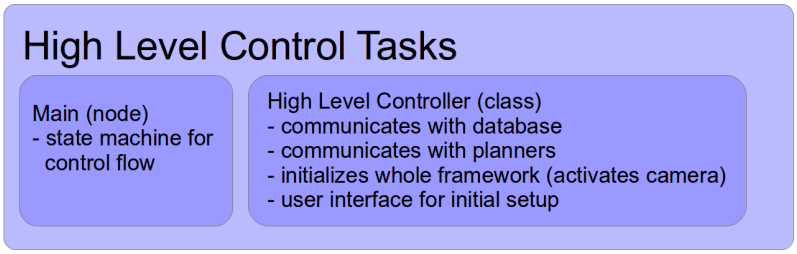


Figure 5.2: High level control tasks.

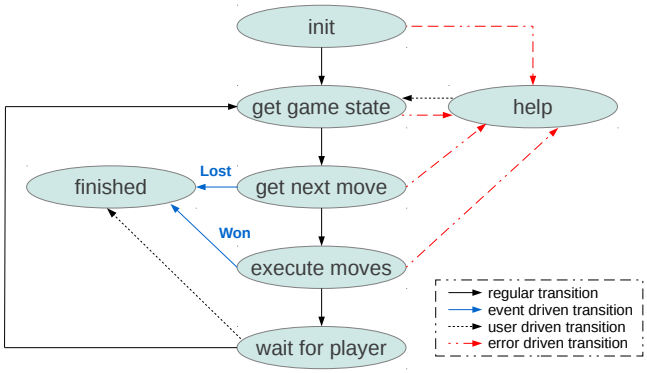


Figure 5.3: Control flow of the overall state machine

1. **init**
   The init state is executed only once during startup. The human player may now decide who starts and if a new game should be started or an old game should be continued. Afterwards the color of both players is initialized. If the player choses to continue an old game, the robot advances to the state "get_game_state". Otherwise it assumes an empty board and tells the player when the initialization is finished. With a touch on the front touch sensor the robot continues to the state "get_next_move".
2. **get_game_state**
   This state updates the complete game state, which is available in the world model afterwards.
3. **get_next_move**
   Depending on which launch file has been started, the database differs. Usually the robot requests the next game state from the database autonomously. However, the framework supports a database simulator that enables the human player in a guided command line user interface to choose the next steps.
4. **execute_moves**
   When the next moves are determined, an "entire_move" (see Section 5.2.1) service request for the planner is built and executed. This step fulfills the complete move of a token including the possible take. After the robot executed all move commands, it continues to the wait state.

5. **wait for player**
   After the tokens have been manipulated, the robot needs to wait for a human player to make a move. The robot takes a resting position and waits until its front sensor is touched. Then the robot changes the state, to acquire the game state.

6. **help**
   In case something fails, the robot changes into the help state. It informs the human player by speech and text. If the player helped the robot by moving it or correcting missing tokens to resolve conflicts, a touch on the middle sensor advances the robot in the ”get_game_state” to begin the next move. In case the player wants to terminate the program, a touch on the rear head button will terminate the high level control loop.

5.2 Planner Package

The planner package is a collection of functions that combine low level functions to behaviors. Additionally the whole kinematics, grasp and walk planning is calculated in this package. An overview of the tasks can be seen in Figure 5.4.

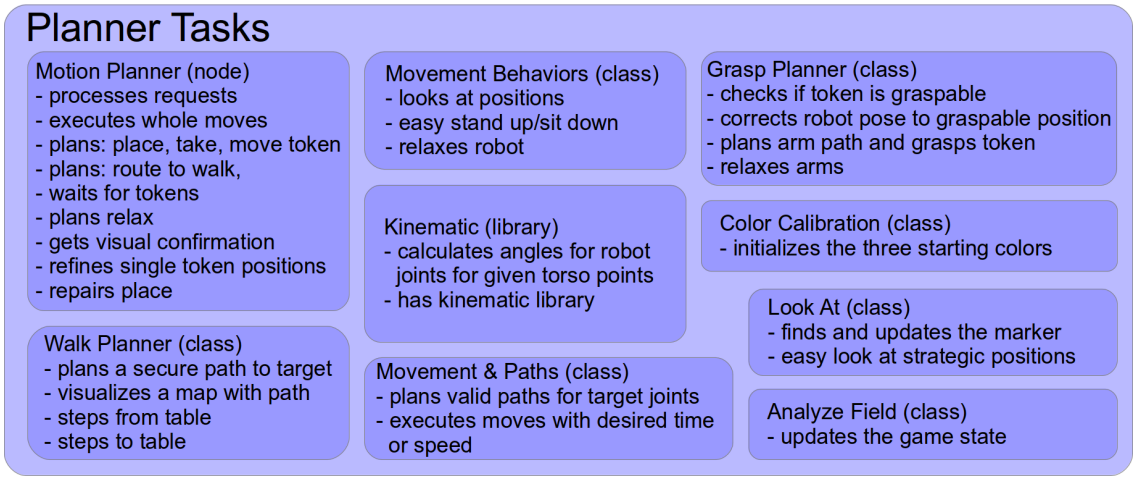


Figure 5.4: Planner tasks

5.2.1 Motion Planner

The motion planner class is the core of the planner package. It offers the “entire move” service, which is a large service that may start a variety of procedures. The service contains a vector of moves. Each move has a type and a vector of double values. The type corresponds to the task ID that tells the motion planner what it should do with the received data vectors. Due to the vector structure an “entire move” can contain multiple moves that are executed consecutively.
The procedure to find the marker requires the time when the marker was last seen. Therefore the planner is subscribed to the marker detection to receive those update times.

Depending on the current move type, the service could execute one of the following tasks:

- starting the game state update procedure.
- starting the color calibration.
- finding the marker.
- executing a place, take and/or move.

A **"place"** requires the player ID and the strategic index where to place the token. If these conditions are valid the robot executes a number of tasks:

1. walking to the closest table side to the target position,
2. finding a torso position to place the token,
3. raising the arm and waiting for someone to hand over a token of the corresponding player ID's color to the robot,
4. closing the fingers,
5. placing the token on the ideal position,
6. retracting and relaxing the used arm,
7. checking if the token is on the expected position, and
8. repairing the place, if it is not on that position.

The **"move"** type request differs from a "place" preceded by a "take", because it requires two strategic positions considering the shortest path and same arm without releasing the token. However, for the first walk procedure only one position is considered in the path planning. Again the robot executes several tasks:

1. walking to the closest table side to the target position, using the same hand to grasp and place,
2. updating the target real grasp position,
3. finding a graspable torso position,
4. grasping the token,
5. raising the hand to a visible position,
6. checking if the token is in the used hand,
7. relaxing the arm without opening the fingers,
8. walking to the second position,
9. finding a torso position to reach the ideal position,
10. moving the hand to the ideal position and opening the fingers, and
11. relaxing the arm.

A **"take"**, could be assumed to be mostly the opposite of a "place" which takes away one token of the board instead of placing it. This is only partially true, because a take is much more complex, because the robot cannot grasp an ideal position. It has to interact with its environment and modify the grasp position. For a take procedure the robot is

1. following the same procedure as the "move" until 6), then it is
2. opening the hand, waiting for someone to take its token, and
3. relaxing the arm again.

The walk to the closest table side includes a prior marker localization to know on which side the robot is currently situated. Afterwards the world model is required to compute the used arm to walk as little as possible (see Section 5.3.3). Then the walk planner computes a trajectory around the board to reach the target destination. Therefore the walk planners also use the visual odometry to support torso rotations.

The repair place method is called, if a placed failed. A flow chart is shown in Figure 5.5. The robot requests a new game state. Usually the lost token should appear in the game state as a token of the player's color. If this is the case, the robot has found the missing token. In case the token has rolled in the area of another token, it does not appear directly in the game state. The vision recognizes a conflict of more than one tokens in a mask of arbitrary color and informs the world model (see Section 5.6.6). Thus the world model is checked for conflicts, if a token is missing.

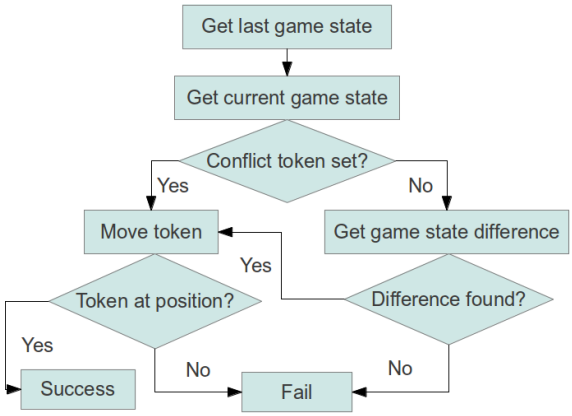


Figure 5.5: Procedure to repair a place action

5.2.2 Analyze Field

The analyze field function fulfills all actions to update the game state. A single update of the game state starts by finding the marker for an exact localization. Afterwards all time stamps are reset, to make all positions interesting again. The decision which position is interesting is evaluated on the time stamp. A loop of the following events is triggered until the world model returns that all positions are updated:

- **activate vision**
  This enables the vision to evaluate one image, which updates the board information in the visible area.
- **get info about update procedure**
  After the vision procedure, the board has been updated in the visible area. Now the next interesting position is requested from the world model. If there are no interesting positions any more, the board is up to date and analyze field is finished.
- **move head to next position**
  If there is still an interesting position, the robot moves its head to the ideal position and the loop starts again by activating the vision.

5.2.3 Color Calibration

This method is used to calibrate the initial color of the game tokens and the blue stripe on the board. For an exact localization the marker has to be found. Afterwards the ideal positions for both players are required. The position 1/0 for the first and 1/2 for the second player, has itself proven to be well-suited init positions. When the robot is looking at the strategic position, the three-dimensional target point is projected into a

pixel of the image plane. The used color is the median color in an area of the computed pixel. The strategic positions can be seen in Figure 1.3. For the blue stripe detection the robot looks at its shoulder, which contains the same color tape as the blue stripe on the board. Additionally some pseudo code for better understanding can be found in Listing 5.1.

```
if(findMarker)
        doForAllPlayers
        {
                lookAtInitPositionOfCurrentPlayer
                pos3D = getIdealPositionOfCurrentPlayer
                pos2D = projectIntoTheImage(pos3D)
                initColorAtImagePoint(2D)
        }
        lookAtShoulder
        initColorAtFixedImagePoint
```

Listing 5.1: ”Pseudo code for initial color calibration procedure”

5.2.4 Kinematics

The kinematics library was developed by Klöbl [28]. He implemented a complete Denavit Hartenberg chain from each foot to each hand using his own written library to compute intersections of lines, planes and spheres. The kinematics are used to calculate the joint angles for a given three-dimensional position. For more details about the calculation, I refer to the work of [28].

5.2.5 Movement and Path

Every time a move is requested, a “movement” object is created and filled in a “path” object. Every “path” may contain multiple “movements”, which are executed consecutively or simultaneously. If a “path” is filled with all required “movements” it can be processed by an ”execute”. The architecture of [28] was designed to cope with a variety of challenges.

5.2.6 Walk Planner

The walk planner is mainly used to plan a safe path around the game board. It uses a potential field algorithm to create a path, as shown in Figure 5.6. The big black structure in the middle is the dilated game board and the blue line is the planned path. The main challenge is to simplify the path to “movement” commands for the robot. This is explained in detail in [28]. Based on the path around the table, it was decided to not walk trajectories with the robot, due to the accuracy and limited working space. Therefore a robot movement, at least for the long path around the table, consists only of straight moves and rotations. A path from the left board side to the top board side, would at least contain two rotations about 90°. One to rotate from, and one to rotate to the board. Most of the time the rotation on the corner of the board consists of two smaller rotations and a forward movement. The accuracy of the rotation is not sufficient enough to walk around the table. Therefore after each rotation that exceeds 45° the visual odometry is called before and after the rotation procedure to measure the rotation deviation and correct it.
Additionally the walk planner is responsible for a safe procedure to step away from the table and to the table. The path planner will fail creating a path close to the table, because the robot is located to close to the dilated board while seating next to it. This prevents plans with trajectories that are too close to obstacles. The procedures to step from and to the table need to be very accurate, otherwise the robot touches the game board.

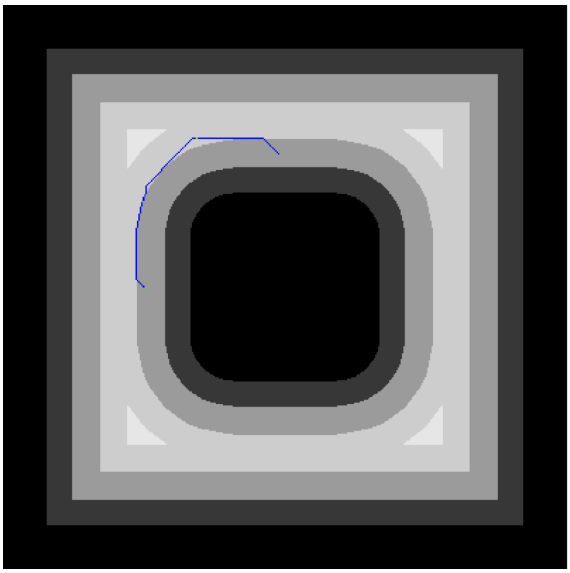


Figure 5.6: The blue line shows the path, where the robot should walk from sector ”LEFT” to sector ”TOP”. The gray levels show the potential field. The darker the color, the more expensive is the position.

5.2.7 Grasp Planner

The grasp planner is used to grasp a specific position on the board, while fulfilling some requirements. The requirements are:

- finding a torso position from which the point is graspable.
- grasping the token from above.
- moving the arm to the position without touching the game board or any other token.

The kinematics only can return a solution, if there is one. If the position is graspable, the kinematics almost instantly returns a valid solution. If it is not, the robot is moved virtually in front of the board, asking the kinematics for valid solution. If a valid solution is found, the robot moves to the graspable position and grasps the token afterwards. On the downside the iterative approach can take a lot of time, depending on the resolution of the evaluated area. Therefore Klöbl [28] precomputed all valid positions for the specific board height with a high resolution. The precomputation for one arm takes approximately 1.5 hours for a resolution with 1 pixel per mm. The precomputed map finds a torso position, where the target token position is graspable. The new position is far away from impossible torso positions, to ensure that the point is graspable. Even if the robot is not exactly on the computed torso position.

5.2.8 Movement Behaviors

This class comprises some movement behaviors for the planner package. It has been implemented to handle frequently used movements in a very simple way. Usually one must keep to the following procedure to move a joint:

1. Transform target pose into torso space, if not already.
2. Retrieve joint angles from the kinematics.
3. Add a “movement” to a “path”.
4. Execute the “path“.

All, except the first step is done by this behavior class, as all behaviors require the target pose in torso space coordinates. The torso is the root of all robot joints.

This class offers the following movement behaviors:

- look at target torso position with desired cam
- set torso height and pitch
- move head around both angles
- relax entire robot

These functions are all very useful and frequently used by the other classes of the planner package.

5.2.9 Look At

”LookAt” is a class that simplifies the head motion. It is mainly used by other classes of this package. One of the core functions is to call the ”lookAtStratPos” function with a given strategic board position. The strategic point is transformed into the ”base_link“ system (see Section 5.3.4) and the movement behaviors are called to move the head to this torso position.

Although the ”lookAtStratPos“ function is useful, the most important and intelligent function of this class hides behind the function ”findMarker”. Due to the complexity to find the marker, a flow chart with the procedure is shown in Figure 5.7. The challenge is that just a positive detection yields a feedback, while the absence of the marker can only be handled using timeouts. So the basic strategy is to check if the marker is visible in the current head position. If it is not, the robot moves the head to the last known marker position, that the marker should be in the middle of the image. The robot waits for some time at each spot for a positive marker feedback. If the robot walked around the table and the walk has finished as expected, the marker should be where it is expected. Sometimes the walk is too inaccurate that the marker cannot be found in the image again in the relocalization step. Then the robot tries to find the marker somewhere in front of its baseline. Therefore the front hemisphere is divided into four parts. The robot makes a quick head move, focusing a rectangle path. This results in a short flare of the marker, if it is there. The exact marker position is still unknown but the quadrant is. As result the robot divides the one quadrant in smaller subparts, looking and waiting at every subpart for the marker detection to give a feedback.
For an exact localization it is important to detect the marker without active motion, as images take some time to be transmitted. If the robot moves, the image is delayed and the localization inaccurate. This inaccuracy may lead to a fail interaction with the board. Especially after a walk, there is only very little tolerance when the robot takes its seating position next to the board. In case the marker still could not be detected, the robot assumes that it is too far away or the marker is occluded and cannot be detected. Therefore the robot also calculated the distance to the blue stripe on the border of the game board, while it was scanning the front hemisphere. Now the robot walks the minimal distance to the blue stripe minus its own size in order not to bump into the board. The strategy is to get a little bit closer to the game board to have a better angle from the camera to the marker. If the blue stripe detection failed or is disabled, the robot does not know what to do, because it has lost its localization and asks for human help. A human touch on the middle head button of the robot continues the program after a ”help” incident.

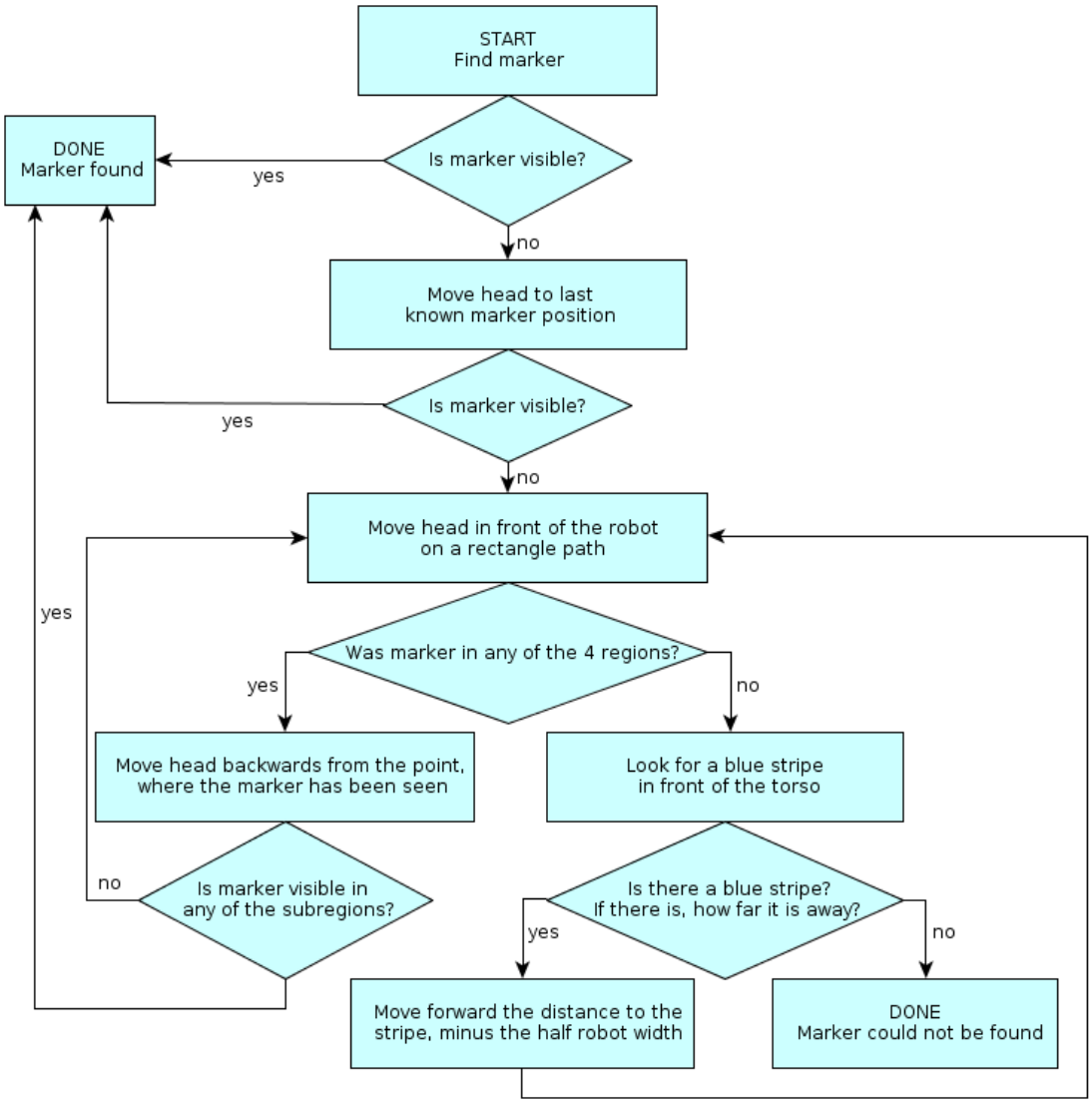


Figure 5.7: Procedure to find the marker

5.3 World Model Package

The world model and knowledge base are base represented by the ROS package ”nao_world_model”. It contains knowledge about the robot, the world and the game situation. Often information is evaluated, fused, or saved in special structures for efficient further use. Every node may request information of the world model, using a service call.
Depending on the task, the world model is split and processed by several classes.

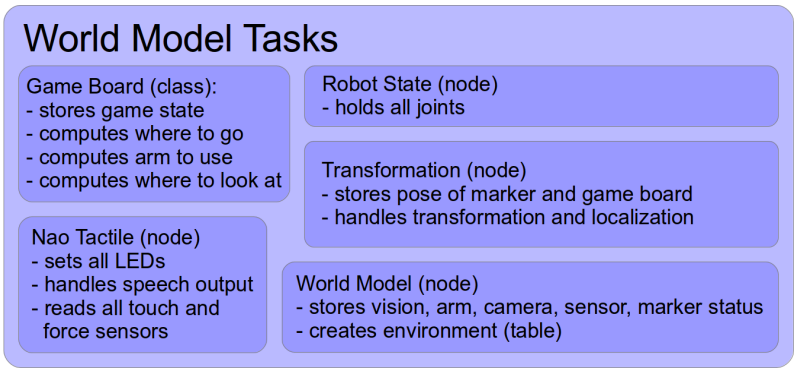


Figure 5.8: World model tasks

5.3.1 World Model Class

This class is responsible to hold the states of the following publishers:

nao_vision_status: gives information about the vision state, for example is it *busy, idle or uninitialized*?

nao_motion_status: gives information about the robot movement.

nao_arm_status: gives information about the used hand.

nao_cam_status: gives information about the current camera usage.

nao_tactile_status: gives information about sensor engagement. This includes the three touch sensors on the head, the chest button, the two bumpers and the force sensors on the bottom of the feet.

The state may be retrieved by a service called ”world_state”, which contains all states. Additionally the world model is responsible for the basic setup, for example the table, ground and an occupancy grid for path planning. Whereas in the initialization the world model calls a service offered by the environment server to set the table and ground objects, the occupancy grid is published periodically by the topic ”map” and is also available by a service called ”world_map”.

5.3.2 Game Board Representation

The game board is being represented internally as a vector of 24 elements for all strategic positions. The first eight elements are the outer square, the next eight elements are responsible for the middle square and the last eight elements represent the inner square. Each strategic position can be accessed by a square and element or index. The same elements of all squares are always collinear. The intersections of all lines on the board, which correspond to the strategic positions are shown in Figure 1.3. The first value represents the square, the second value the element. For instance, if we consider the Nine Men’s Morris board from the last mentioned figure, then the bottom left position of the outer square would be (0,0). The bottom right would be (0,2). The same position on the inner square would be (2,2). This way each of the 24 positions is represented. The strategic positions could have one of the following occupations:

0: The position is empty.

1: The position is occupied by player 1.

2: The position is occupied by player 2.

3: The position has too little support to say anything about it. The occupation is unknown.

5.3.3 Game Board Class

The base specifications of the game board were already described in 2.3. In this section the practical implementation of the game board class is described. The game board class ”GameBoard.cpp” can be found in ”nao_world_model/src”. It serves as the brain of the board as it holds

- the board occupation and real positions of all tokens.
- the ideal positions on the board.
- the current sector of the robot in relation to the board.
- interesting positions, if an ”update game state“ is requested.

Therefore it offers the service ”game_board_task” for other nodes to provide the recently mentioned information. The first parameter of the service request is the ”task_id” that defines which subfunction is called. If the service is running, a status is returned.

The service offers different tasks which can be called with a request of the correct task id.

task_id == 0: getIdealPositions: In the game board all ideal positions are defined. If any node wants to know these positions, the game board offers a service that returns the ideal positions in any requested coordinate system. The desired coordinate system is defined in the service request. After the service response has been constructed, a vector with 24 points is returned together with a status flag. In the case that an invalid system name is requested, the service returns an empty vector with a negative status flag.

task_id == 1: isWalkRequired: This service is called, if a token should be grasped or released. It tells the demanding node to which table side the robot has to move to grasp the requested token. For this request the strategic element number, from 0 to 7, and the arm index(0, 1 or 2) are required. The square is not required, because the specific element in all squares is graspable from the same table side. An arm index of 0 means that both arms are free for use, whereas an arm index of 1 tells the algorithm that the left arm has to be used. We define that nine positions have to be graspable from each side, which are three positions in each square on each board side. The three positions on the corners of each side are reachable from two different sectors. The current sector is calculated by the angle around the vertical game board axis (z-axis) that is obtained of the transformation of the torso in the marker system. With the current sector, the element and arm index, the minimal required movement can be calculated. The arm index holds the information to use a specific arm in case a token is already grasped by a specific hand. The positions in the middle of each board side that correspond to an odd element value, are always considered to be graspable with both hands but only from one sector. If a an odd element has to be grasped, the sector is already determined and the arm does not matter. Tokens on corner positions are also graspable with both hands but different torso positions and sectors. In case there are no further arm restrictions, the sector with the minimal walking distance is chosen. Algorithm 1 shows an example of the decision making process for all sectors and elements. The formula in this algorithm maps every even element value to two possible sectors. The sector with the lower costs is chosen, if there is no arm restriction. The corresponding arm is also known after the sector with the lower costs has been chosen. Finally the sector where the robot should move and the arm which should be used, are returned by this algorithm.

task_id == 2: updateGameState: This service call requests the most interesting position to look at in the marker coordinate system and a number how many positions still need to be updated. Until this update value reaches ”0”, the board is not updated. The most interesting position is defined by the oldest time stamp. So all strategic points are examined and the position of the oldest point is processed further. The strategic point is transformed into offset values from the marker. If the head is moved to another position, all time stamps of the ideal positions that are visible, are updated by the vision. If the last update time of all strategic positions is below a threshold, the game state counts as updated.

Algorithm 1: calculate best destination sector and arm

input : ele ... element number [0-7],
a ... arm index [both(0), left(1), right(2)],
sec_{from} ... sector of the robot [0-3]
output: a ... arm index [both, left, right],
sec_{to} ... sector to go [0-3]

if $odd(ele)$ **then**
  $sec_{to} = \frac{ele-1}{2}$
else
  $sec[1] = \frac{ele}{2}$
  $sec[2] = \left(\frac{ele}{2} - 1\right) \%4$
  $sec_{to} = min((sec_1 - sec_{from})\%4, (sec_2 - sec_{from})\%4)$
  if $sec_1 == sec_{to}$ **then**
    $a_{tmp} = 1$
  else
    $a_{tmp} = 2$
  end
  if $a! = a_{tmp}$ **then**
    $sec_{to} = sec[a_{tmp}\%2 + 1]$
  else
    $a = a_{tmp}$
  end
end

task_id == 3: getGameState: It is only recommended to call this service after the update value of the call ”updateGameState” returned ”0”, because only then all positions have been updated and the updated game state may be retrieved. The current game state is returned as a vector of integers with 24 elements that contain the occupation. Additionally to the strategic occupation the three-dimensional positions in marker coordinates are returned. If this task is requested before the vision has updated the board information the first time, a warning is produced or the service will fail.

task_id == 4: getRobotSector: This task returns the current sector of the robot in relation to the marker.

task_id == 5: getConflictToken: This task returns the index of the conflicted token and a three-dimensional position for that index in the marker system. This task is used to repair a failed place.

The game board requires periodically updated ”board information“ and therefore it is subscribed to the topic ”boardInfo”. This message is published by the vision. It contains updated information of the visible token positions. Furthermore the ”boardInfo” has a status flag that is ”0” if the ”token information“ is valid or negative otherwise. Only a valid ”board information“ updates the world model. Occasionally it contains one additional token position. In case of a place failure this ”conflict token” contains a three-dimensional position and strategic index. This serves as a fast check and repair mechanism, which is discussed in Section 5.2.1 of the planner, whereas the conflict tokens are described in Section 5.6.6.

Additionally the game board uses the transformation stream that also is running in the world model and is described in detail in 5.3.4. The transformations are required to transform the defined marker points into points of the world, camera or any other system.

5.3.4 Transformations and Localization

Usually robots consist of multiple modules. Hands, arms, legs, movable platforms, laser scanners or cameras. These components do not know each other. The sensors (for example: cameras) offer and the actuators (for example: motors) require their information only in their local coordinate systems. Therefore the data has to be transformed between the coordinate systems, which makes the data management and analysis much easier. For example, a grasp position is always given in coordinates relative to the torso.

In contrast to a projective transformation, the coordinate transformation, which is required to transform robot frames only, consists of a maximum of three translations and three rotations in a three-dimensional space. Each translation and rotation can be defined by one parameter. A sample translation matrix for all directions in three-dimensional space in homogeneous coordinates looks like Equation 5.1. The translation in all three directions are located in the last column of the matrix.

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

Whereas three rotation matrices, shown in Equation 5.2, are each responsible for one rotation around an axis. ϕ denotes the angle around the x-axis, ψ the angle round the y-axis and θ the angle around the z-axis.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi & -sin\phi & 0 \\ 0 & sin\phi & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_y = \begin{bmatrix} cos\psi & 0 & -sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ sin\psi & 0 & cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

Those four transformation matrices can be combined into one matrix by a simple matrix multiplication in the correct order, like $P = R_z \cdot R_y \cdot R_x \cdot T$. The order is important, because matrix multiplications are not commutative. A point can be simply transformed into the other system by a multiplication with the transformation matrix, for example $x' = P \cdot x$. The same applies to a system with an orientation.

It not necessary to transform all data manually, because ROS offers a transformation service called ”tf”. In order to use the tf-service, the some transformations have to be provided. The tf-stream is similar to the publisher-subscriber communication with additional functions. There are several nodes that broadcast transformations to the tf-stream. One transformation contains the relation between two systems, their names and a time stamp. The transformation relation contains three parameters for translation and three or four angles for rotation (see Section 5.3.4 for Quaternions). More provided transformations automatically build a tree shaped structure, if the system names (frame_id) fit together. Every ROS node is able to create a tf-listener to use all transformations that are available in the tree. Static transformations do not change their relation over time in contrast to other transformations that need to be updated periodically. Once the tf-stream is provided with the correct transformation between two systems, any pose between any two systems of the transformation tree can be transformed. For instance one transformation from system A to B and one from B to C is published. Then a point can be immediately be transformed from the A in the C system.

We decided to have all movable joints of the robot in the tf, because it is easier to work in each joints base coordinate system. We visualized the tf-tree with all systems that are used in the project in Figure 5.9. However the arm and leg systems are summarized due to the amount of links. These systems are shown in a sub-tf-tree in Figure 5.10. Note that the origin or root of the tree is the world that serves as parent for the marker ”(odom_combined)“ and the torso ”(base_link)“ frame.

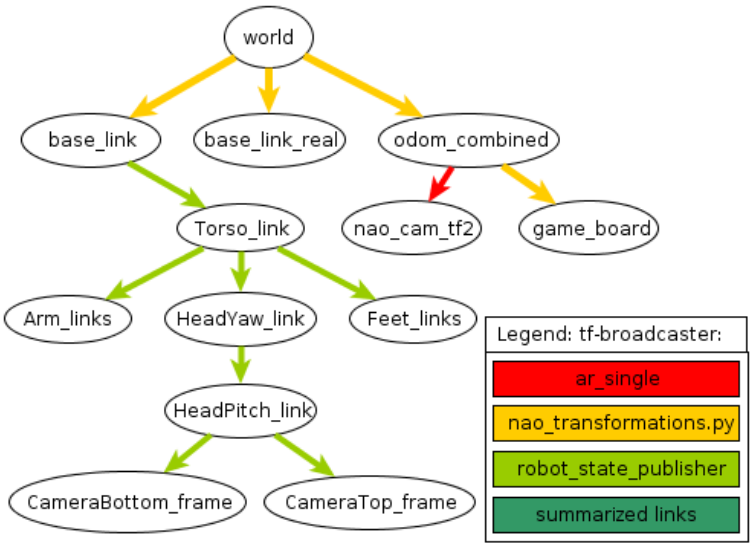


Figure 5.9: The transformation tree of all coordinate systems in the project with summarized arms and legs.

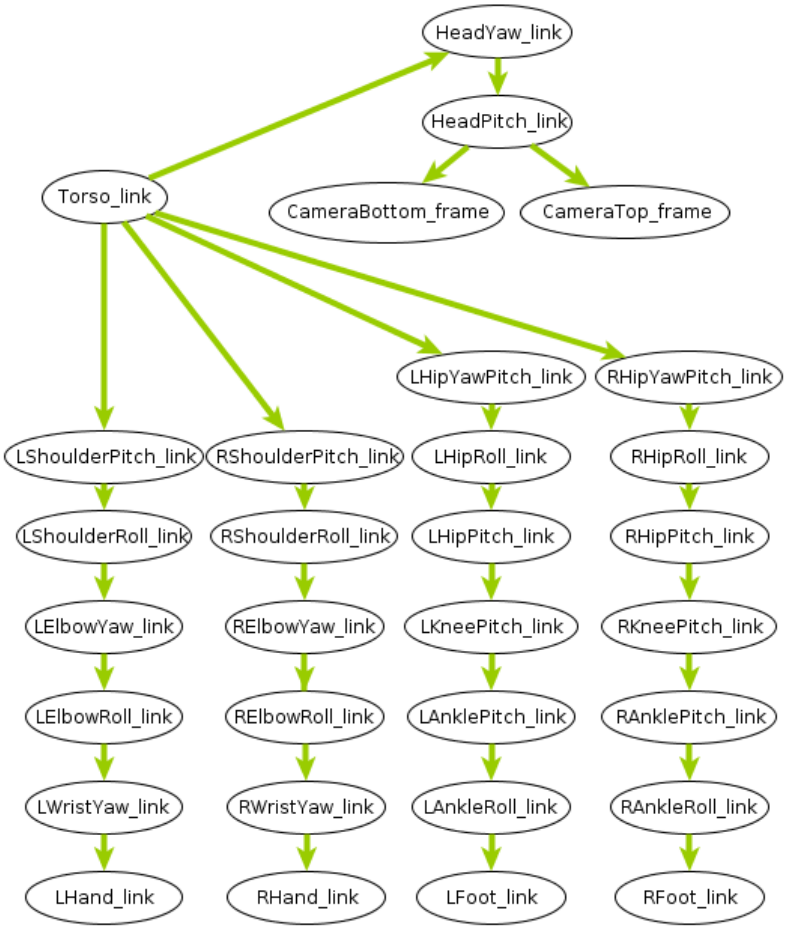


Figure 5.10: The transformation tree of the robot containing legs, arms, head and cameras in relation to torso.

In fact a tree must not contain circles to avoid that transformations are not unique any more. This also leads to a challenge in this project, because there are three sources that estimate the torso position. (1) The odometry measures motion differences, but is greatly influenced by the underground, obstacles or bumps on the path. (2) The transformation of the marker detection is more accurate than the odometry, but not available, if the marker is not in the image or it is too far away. (3) The visual odometry can only manipulate the torso rotation around the vertical z-axis of the robot. Due to uncertainties and drawbacks in all systems, none of them is reliable. The camera provides information about its relative position to the marker, if it is visible. In case the marker is not visible, the robot is still able to move around. Then the odometry information of the robot itself has to be used. Some experiments showed that the position in relation to the marker, gained from the marker detection of the camera, is often more accurate than from odometry, regarding the x-y position. This is based on the fact that the odometry accumulates errors in contrast to the marker detection, which calculates the absolute position each time.

In experiments, an issue with the marker was discovered, when the inner square is occupied by tokens and the robot is not standing next to the table. In this case the view angle and the tokens are so high that the marker is occluded by the tokens and therefore it cannot be found any more. The solution was to mount the marker on a wooden plate in the size of 70x70x12mm to improve the detection, as explained in Section 5.6.2. The drawback of this modification is that the inner tokens are now a little bit trickier to grasp.

Until now the ideal game board positions were at the same height as the marker itself. A new system ”game_board” is introduced as a ”child” of the marker system that is called ”odom_combined”. The relation of these systems is a static transformation of the height of the marker above the upper table side. Due to a node, which is already implemented bye ROS, called ”planning_scene”, the names ”odom_combined” and ”base_link” are fixed. All combined transformations automatically lead to a localization of the robot in the world.

In Figure 5.11 the basic systems that are required for localization, are shown on the left and the corresponding camera image on the right. The red line always represents the x-axis, the green line the y-axis and the blue one stands for the z-axis. The forward movement of the robot is always in the direction of its x-axis in the robot coordinate system, whereas the visual line of the camera is the z-axis in its coordinate system. This is a very important fact, if a transformation lookup between the torso and the camera is requested. In this case it is challenging to imagine the retrieved rotations. Nevertheless transformations facilitate the handling of tasks with multiple systems.

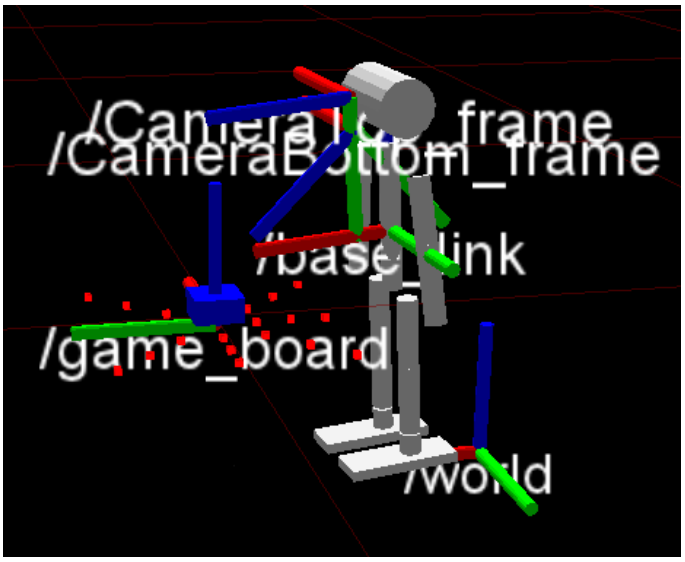
There are currently three Python scripts that are updating the transformation stream of ROS periodically.

1. nao_tf.py
2. ar_single
3. robot_state_publisher

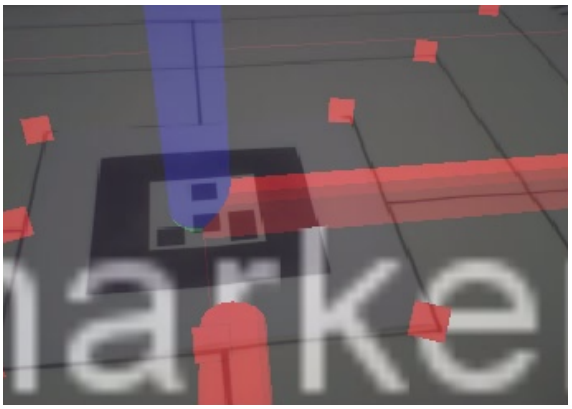
The first script ”nao_tf.py“ is responsible for the transformations between the robot and its environment. The second script only publishes the marker to camera transformation, as it is based on the ”ar_pose“ package and the third publishes all joints of the robot. As already shown in Figure 5.9, the transformation relations are color coded to visualize their source node that is updating the transformation. The second tf-tree in Figure 5.10 only shows robot joints published by the ”robot_state_publisher“.

The most important transformations, concerning the localization, are listed below.

- **world - odom_combined**
  In this static transformation the marker is always at the same location in relation to ”world“. This is an offset considered from ”world“ of the x-axis of 40 cm and 23 cm of the z-axis and a rotation around the z-axis by $-\frac{\pi}{2}$.
- **odom_combined - game_board**
  For this static transformation, the ”game_board“ has the same x-y-position as ”odom_combined“ but is 12 mm lower.



(a) Coordinate system representation



(b) Internal camera view

Figure 5.11: Figure 5.11a represents the most important axes of the coordinate systems used for the localization. Figure 5.11b shows the camera view to the marker. The small red squares represent ideal positions on the game board.

- **world - base_link**
  This transformation is the most complex transformation in this project. It is responsible for the localization of the robot in its environment. Usually it is updated by the odometry of the NaoQi. Sometimes the robot is moved by the human without restarting the NaoQi, so the position of the robot is changed. Although this should be updated in the NaoQi, it is not possible to update the position in the NaoQi by external sources. As a result, we create our own coordinate systems which allow a sudden change of the robot pose. The NaoQi provides only an absolute positions, even though only the lastest pose changes are required. Therefore the difference from the last to the current pose is calculated. If there is a change, we add this to our last position. Like all odometry systems, this leads to an accumulation error and needs to be corrected. At this point the marker is involved. If the robot requires a more exact localization, it requests a marker update. So the robot is looking for a marker. If a marker is found, it publishes its transformation from the marker to the active camera. However the active camera has a different system name ”(nao_cam_tf2)“, because otherwise the odometry and marker create a circle relation, which is strictly prohibited. The marker to camera relation is transformed into the ”base_link“ system. As a matter of fact this position is absolute and

overwrites the last ”base_link“ position. As long as the marker update is enabled the right green eye LEDs are activated to visualize the update procedure. Afterwards the green eye LEDs vanish and the left red eye LEDs are active again to show odometry update.

Additionally there is a service to correct the robots rotation (yaw) externally. This is required for a walk around the board. Sometimes the odometry error is higher than 10° of a 90° rotation. This is not acceptable, because it leads to a false position of the robot. Therefore a visual odometry has been implemented to correct the rotation deviation, which is explained in Section 5.6.8. The visual odometry measures the difference of the expected rotation and the real rotation. If the deviation is too high the robot corrects it by rotating around the difference angle. This correction angle has to be ignored in the localization of the world model, because the robot already assumes to be orientated correctly.

- **base_link - base_link_real**
  The ”base_link_real“ system is used for better grasp positions. It is a fusion of the marker detection and the odometry. While the ”base_link“ contains the pure marker transformation, if it is visible, the ”base_link_real“ gets its height and pitch from the odometry.

The periodic transformation updates are required by the visualization tool ”rviz” and other implemented nodes (see Section 5.9), like the game board or head motion that use transformations. The other transformations that contain all joints of the robot are mainly required for kinematics and motion planning. These transformations are computed from the URDF model, created by Klöbl [28]. Additionally some objects, like the table and a ground, are simulated. The robot joints and the table are required to define constraints, to tell the robot the forbidden zones, where there would be a collision with one of its body parts and the table.

Euler Angles and Quaternions

ROS stores and computes all rotations of the transformations in quaternions, which require four angles/parameters. The tf also offers easy methods to transform between Euler angles and quaternions.

The following section is based on the work of Bartz [6]. Quaternions were invented by the Irish mathematician William Rowan Hamilton. He searched for for an extension of complex numbers in the three-dimensional space. Therefore he had to define a forth-dimension to multiply the complex numbers. However, it was inconvenient to use quaternions until Josiah Willard Gibbs defined that quaternions have to consist of a scalar and vector part, which led to a serious simplification.
Similar to the rules of the complex numbers the square of each of the three imaginary numbers results in minus one. In contrast to normal complex numbers, the multiplication of the imaginary parts is not commutative. This resembles the multiplication of rotation matrices, which is also not commutative.
Advantages of quaternions versus Euler angles are:

- Concatenations of rotations in quaternions are more efficient as no 3×3 matrices have to be multiplied. Instead of 27 multiplications and 18 additions, only 8 multiplications and 4 divisions are required.
- It is not intuitive to retrieve all Euler angles to a destination, as the sequence of the rotations not commutative.
- The major reason why quaternions are used, is the so called ”gimbal lock”, which happens on a specific rotation with Euler angles. This happens if one plane is rotated in another plane, leading to a loss of a degree of freedom. If another point should be reached the coordinate system now performs a random trajectory. In the quaternion system the destination point can always be directly reached.
- One cannot conclude the basic rotation out of an Euler rotation matrix.

Disadvantages of quaternions:

- Quaternions are not basic mathematics and therefore rather unknown to the public.
- The matrix representation of Euler rotations can additionally be used for translations, scaling and projections.
- Only rotations can be computed with quaternions, which requires a conversion for coordinate transformation.

More detailed information can be found in [13].

5.3.5 NaoTactile Class

The class ”NaoTactile“ is responsible to manage the robot's sensors, speech and LEDs using library functions of the NaoQi. These functions require one subscriber and publisher.
The subscriber receives messages, which contain a status and a string. If the status is *zero* the LED that corresponds to the string is switched off. The status *one* switches the corresponding LED on. Any other status is treated as text to speech output that synthesizes the string.
The publisher is responsible for all head buttons, the chest button and bumpers. These functions are triggered by events and result in a published sensor status change. Additionally this node watches the force sensors on the feet, but in contrast to the other sensors that are triggered by events, the sensor values need to be requested periodically, since there is no working event function. A change in the sensor values that is greater a threshold, results in a status change and is published too.

5.4 Database Package

The database package contains the game engine framework of Regenfelder [34], which communicates with the Nine Men's Morris database of Staber. The database provides the game intelligence for the robot, as it contains the next best k-moves for all states. The final database has an expected size of about 100 GB. For current flash memory this too large, as the robot currently has only an equipped memory stick of 2 GB. Thus we decided to use an external notebook that was connected by a network cable to compute the required data. It is also more convenient during the development process to have a remote access to the robot. Figure 5.12 gives an overview of the tasks of the database framework.

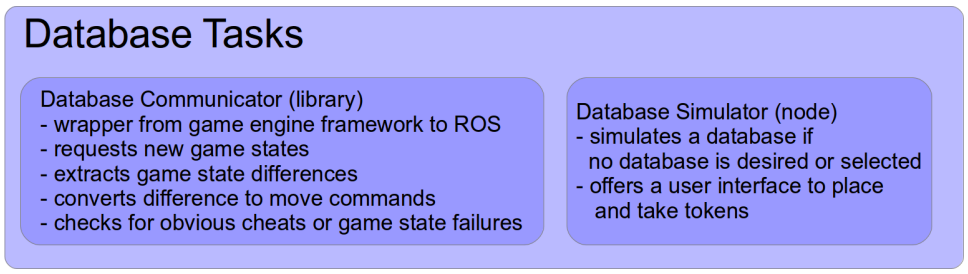


Figure 5.12: Database tasks

The ”DatabaseCommunicator”-library serves as an interface between the game engine framework and a ROS service for our framework. On a new request the service takes the current game state and returns the next move. As the database uses another representation of the game board, all required information has to be converted. A request to the database requires a string with 27 elements. This string consists of

- 24 elements to represent the current state,
- 1 element for the current player ID, and
- 2 elements to represent the game phase.

The 24 elements of the database represent the strategic positions from top left to bottom right, line by line in contrast to our square and element representation. Therefore a conversion of the game state has to be applied. The description of the meanings of all elements can be found in Table 5.1.

Element description	Nao	Database
Empty position	0	’O’
Player 1 token	1	’B’
Player 2 token	2	’W’
Player 1 turn	1	’A’
Player 2 turn	2	’B’

Table 5.1: Conversion table of game state elements

Additionally the game phase is required to get the next game state, as described in Section 4.2, the movement type of the tokens changes with the game phase. It could not be recovered from the game state only. In the first game phase, the tokens have to be placed. Hence the encoding from ’00’ to ’08’ corresponds the number of placed tokens. The game phase encoding is shown in Table 5.2.

Game phase	String encoding
Initial placement	’00’ to ’08’
Move phase	’09’
Jump phase	’10’

Table 5.2: Game phase encoding

The robot is always the current player, except for the cheat detection, which should be applied to moves of the opponent. The question whether the robot is player one or two depends on the initialization. After the conversion and a request to the database, a string with the next best game state is returned. This string is converted back to a ”Nao game state” for more efficient processing. The old and the new state are analyzed and the differences are saved. With these differences move commands are created for the robot. These move commands consist of an edit option that contains the encoded actions (Table 5.3) and an index vector with three elements. There are only four possible edit options as no ”take“ is possible without placing or moving a token.

	Integer encoding	
Edit type	decimal	binary
Place	1	001
Move	2	010
Place & Take	5	101
Move & Take	6	110

Table 5.3: Edit option encoding

Depending on the edit option, the three index elements are processed consecutively and contain

1. the token position to place or move from,
2. the token position to move to, and
3. the token position where to take away an opponent token.

So the service of the database library returns

- the edit option,
- a vector with indices to manipulate,
- an value with the maximum number of steps until the game is over, and
- the classification (win, lose or draw game).

5.4.1 Database Simulator

If the database is used, it tells the robot, what it should do next. Sometimes the user want to tell Nao to place or grasp a specific position or walk around the board. Hence we implemented a simulator, which allows a human to fulfill the task of the database by choosing Nao's next moves. The user decides the position, where a token should be placed or removed. Thus, the user is able to remove up to two tokens and place one per step. The move commands are automatically created by the game state difference. For example, by removing and placing a token of the robots color, the robot interprets the change as a move command of one of its tokens to another position. The user is informed about invalid entries.

5.4.2 Cheat Detection

A simple cheat detection was also implemented that can be enabled at startup. Until the move phase, the number of tokens of each player is verified with the game phase, considering lost tokens due to a mill. In the move phase additionally invalid moves that move more than one position also considering a token loss, are exposed.
In future version all inconsistencies in the game state should be found, using prior knowledge (game phase and prior game states).

5.5 Helper Package

Originally this package was created, because messages and services were required by multiple packages. These package should not include themselves mutually or should not have a dependency at all. Now it comprises a collection of very useful functions and definitions. Figure 5.13 shows a summary of its core functions.

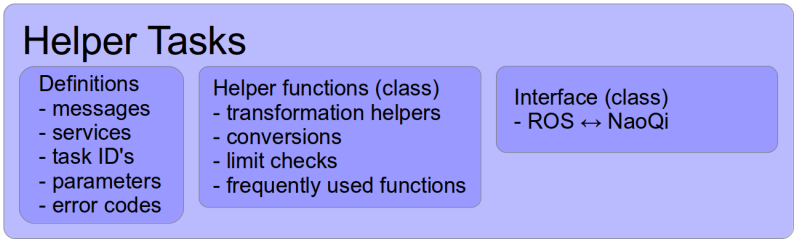


Figure 5.13: Helper tasks

5.5.1 Interface: NaoQi - ROS

Aldebaran states that the programming language C++, Python and Urbi are supported. We chose Python as communication language to the robot. In contrast to C++, Python does not require a boost library which makes this approach more robust against updates. Additionally Python seemed to be most used language for programming Nao.
The communication with the robot requires some libraries, also called NaoQi, that is available at the Aldebaran-Robotics website [2], with a valid user account. The NaoQi version 1.12.3 for a Linux with 32 bit was used. A 64 bit Linux was not used, because the drivers for the marker detection are also only available for 32 bit at this very moment. As a first step an environment variable "PYTHONPATH" needs to be set. The set variable should include the libraries of ROS and the NaoQi.

$PYTHONPATH = "/pathToRos/ros/electric/ros/core/roslib/src:/pathToNaoQi/naoqi-sdk-1.12.3-linux32/lib"

This variable is required to import the NaoQi in a Python script. Afterwards a package and a Python script to access the NaoQi and ROS, can be created. The first lines should look similar to the example code in Listing 5.2.

```
try:
        from naoqi import ALProxy
except ImportError:
        print("set your pythonpath!")
        exit(1)
import rospy
```

Listing 5.2: "Example code to import the NaoQi library and Python for ROS."

ALProxy is a helper to retrieve nearly all libraries to access the robot. Only ALBroker for the sensors needs to be imported the same way as ALProxy. The access to another proxy with the help of ALProxy requires the name of the requested proxy, the IP address and the port of the robot. It is also possible to run a local simulation. Therefore the local IP address (127.0.0.1) is required. Aldebaran did not provide all modules for simulation, for example the proxies for the LEDs and speech.

In the line below there is a sample that shows how to retrieve the motion proxy of the robot, which is used to control every movement of the robot:

```
motion_proxy = ALProxy("ALMotion", robot_ip, robot_port)
```

A complete list of methods for each proxy can be found in the documentation on the CD that comes with the robot, online [4] or on the robot itself. By entering the IP address in a browser, all modules that are loaded and active can be seen. Sensor values and their subscribers are also visible in the interface. Additionally it offers some simple preferences of the robot to control the spoken language, the volume level and other basic preferences.

In the framework the connection script, called "naoqi_nao_node.py", can be found in the source directory of the helper package. This script offers some functions to easily retrieve proxies. The motion script "nao_motion_control.py" is the interface to movement commands. The vision script "nao_cam.py" subscribes to the cameras, the world model scripts "nao_tf.py" handles the transformations and localization and "robot_state.py" checks the joints. Furthermore there is another script in the world model "nao_get_data.py", which checks the touch and force sensors and also offers an interface to control the LEDs and speech. All these scripts use the connection script to obtain their proxies.

5.5.2 Definitions

The helper package is the place for framework definitions. Currently the definitions contain:

- all defined message types
- all defined service types
- error codes
  An error code is an integer value that consists of the package name, the node name and the error code itself. This is very helpful for debugging, because the developer knows instantly where the error occurred. Every time a function returns a negative value, something went wrong. A return value of zero states that everything is fine.
- task ID's
  Task ids are required by services to distinct tasks.
- parameters
  Definitions of adjustable parameters are defined, for instance board size, camera resolution, arm and player index and sectors.

5.5.3 Helper Functions

The package offers useful functions, for example a transformation helper that contains an updated listener and has a method for an easy transformation. Additionally conversion methods and often used functions are implemented in this package, for example "sub2ind", "vec2point and "str2int". They are used to convert strings, integers, vectors, points and others. Furthermore there is an output helper that formats the console output. Instead of the ROS time, the output helper logs the package thats posts a message. The complete color range is also available to color messages for a better visualization. Moreover all outputs of every node of our framework can be set in one central configuration file for different debug levels.

5.6 Vision Package

The vision package deals with all tasks that include the camera image. The results of the vision are listed below as it offers

- a structure that contains information about every of the 24 strategic positions. This information comprises the discrete occupation, the real positions, if they are occupied and time stamps of the last update. Additionally the structure contains a 25^{th} token that is only set, if there is a conflict on the board. This so called conflict token, has the same information as any other token plus an index to the corresponding strategic position. The index is important, because usually the position of the vector element is implicitly the index of the token. The conflict token marks a mask or strategic position, where two tokens of arbitrary color are present.
- a rotation angle for the vertical torso rotation, computed by the visual odometry.
- a confirmation, if a token is in a desired mask.

The vision only requires the camera image, the camera height and inclination to the world to achieve these tasks.

The submodules and their duties are shown in 5.14

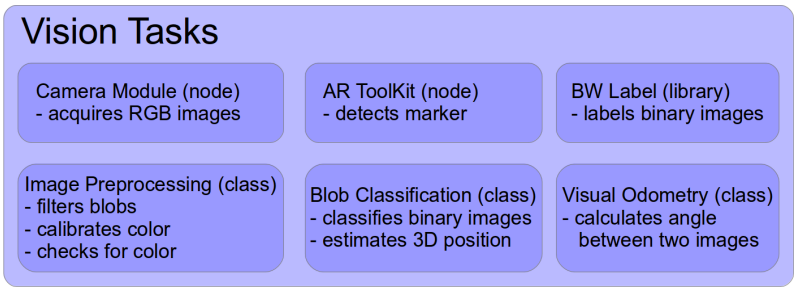


Figure 5.14: Vision tasks

5.6.1 Camera Module - Image acquisition

The major ROS node for the vision is the node called ”nao_cam.py” in ”nao_vision/scripts”. This node is responsible to get the current image from the NaoQi. Nearly all vision nodes depend on this node implicitly, because the image is their primary input. The core of the ”nao_cam.py” node is the generic video module subscriber to one of the cameras with 10 frames per second. It receives the last image of the camera with a resolution of 320x240 in RGB color space and publishes an image about every 100 ms as long as the robot is stationary. An example code to connect to the camera is shown in Listing 5.3.

```
GVM_name = "nao_cam"
resolution = kQVGA #320x240
color_space = kRGBColorSpace
fps = 10
cam_state = 1 # bottom_camera = 1, top_camera = 0
# requires previously imported ALProxy
cameraProxy = self.getProxy("ALVideoDevice")
GVM_name = cameraProxy.subscribe(GVM_name, resolution, color_space, fps)
cameraProxy.setParam(kCameraSelectID, cam_state)
image = cameraProxy.getImageRemote(self.GVM_name)
cameraProxy.unsubscribe(self.GVM_name)
```

Listing 5.3: ”Example code to get an image from the Nao camera”

During motion the images are blurred and therefore no precise localization of the robot and the tokens is possible. However, the images are published continuously, because the marker detection node needs an updated image to recognize when the marker is lost. A special feature of this node is to swap the camera. Although due to hardware limitations, it is not possible to use both cameras simultaneously, the robot may require the top camera for localization. Therefore the node offers a service to switch the camera.

The bottom camera is mainly used during game play. It offers a well-suited angle to see the game board in front of the robot, while the top camera is better suited in walk procedures around the table. During walks, the bottom camera only sees the white table, while the top camera in the horizontal, is able to observe distant objects. Hence the top camera image yields much more interesting features in the images, which is important for localization.

If the camera state changes, the camera is switched and the *frame_id* in the header is updated with the new camera *frame_id*. The *frame_id* is important for ROS to organize different coordinate systems. This is very important for the transformation and for the visualization in rviz to arrange the camera’s position in the scene depending on its *frame_id*. In this case the *frame_id* is ”CameraBottom_frame” for the bottom camera or ”CameraTop_frame” for the top camera. In addition the transformation for the new camera has to be updated by the transformation script ”nao_tf.py” in the world model.

5.6.2 AR ToolKit

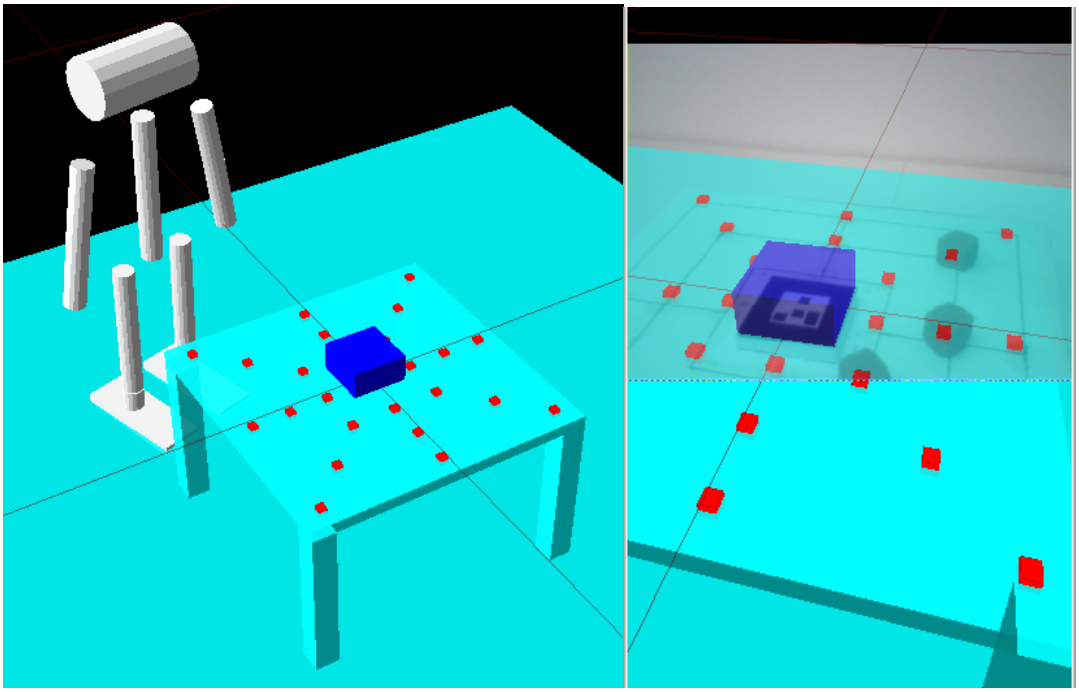
As an exact localization was required, a marker was used that can be detected by the AR ToolKit [5] using the internal camera. According to the work of Zhang et al. [45], where different markers were compared and evaluated, the AR ToolKit yielded the best trade-off in recognition time, deviations and projective distortions. Furthermore a free and non-commercial version of the framework is available online. The AR ToolKit developed by Kato and Billinghurst [26] is a software library for augmented reality. It is used to track a real world marker to estimate the pose of the camera in real-time. Therefore only a single calibrated camera is necessary in combination with a known square marker pattern. The estimated camera pose (position and orientation) is required for self-localization of the robot. Another reason to use this framework is an already existing wrapper for ROS, which was very useful to integrate the detection in our framework.

Due to the inaccurate odometry of the robot while turning, it would be a challenging task to move the robot around the table without any visual sensor to correct and support the robot. If the robot walks around the table, it waits before approaching to the board again. At first it tries to relocate the marker. After the walk with rotations, the odometry estimation of the robots position is somewhat inaccurate. Due to the absolute transformation, the AR ToolKit is much more accurate. This accuracy is required to safely approach to the game board.

The camera pose to the marker is most accurate, if it is watched frontally with a perpendicular angle from the camera center to the center of the plane of the marker. If the view point angle is reduced, the transformation becomes more inaccurate, until the complete detection vanishes. According to the documentation of the AR ToolKit, the marker detection for a marker with a side length of 70 mm is not guaranteed over 40.7 cm in a perpendicular view angle. A perpendicular view angle is not possible due to the position of the marker and the robot that is situated on the side of the table, as shown in Figure 5.15 and 5.16. In this Figure, the robot has an angle of about 60° with a distance to the marker of about 40 cm. Assuming the robot moved backwards, all conditions are impairing. The inclination angle to the marker increases, so does the distance. Additionally the view to the marker, caused by the altered inclination angle, could also be occluded by game tokens.

Due to the severe occlusion by game tokens, the marker has been raised by 12 mm. We did not raise the marker to the same height of the tokens to completely eliminate occlusion, because two new issues would occur. Firstly the inclination angle is increased with the height of the marker. Secondly, if the marker has the same height as the tokens, the robot cannot place a token in the inner square any more, because its fingers are too thick and the marker would block them. This height is a trade-off. The robot may not see the marker from behind, but when it steps closer to the board the marker becomes visible again.

Each used marker pattern has to be saved as a template once. There is already an executable, called ”mk_patt”, in the ”bin” directory of the AR ToolKit, which creates template patterns for markers using a camera. The correct transformation to the marker can be obtained, after the size of the marker has been adjusted in the vision package.



(a) Rviz representation (b) Robot view with rviz overlay

Figure 5.15: The robot's position where the picture of Figure 5.15b was taken, can be seen in the world model in Figure 5.15a, which has been visualized in rviz with the robot model from [23]. Both figures show the detected marker in Nao's world. The external view of this scene is shown in Figure 5.16.

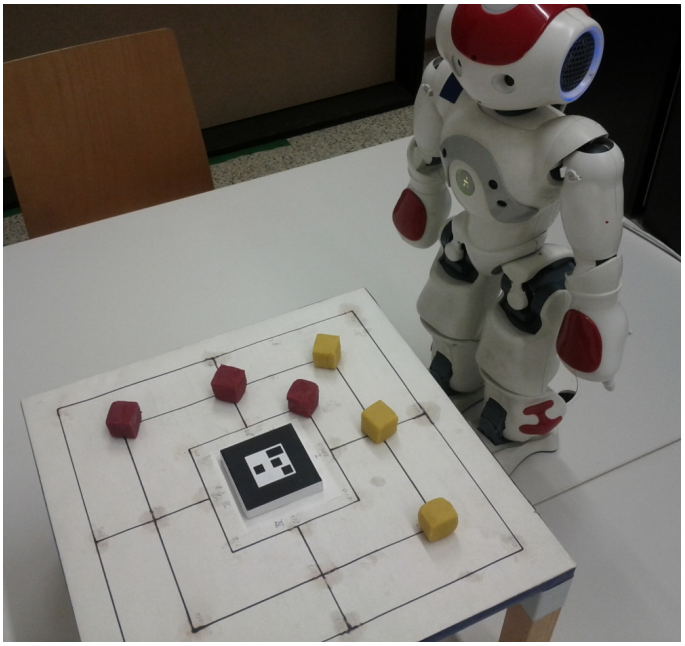


Figure 5.16: This image was taken by an external camera and shows the robot pose in relation to the marker, which corresponds to the setup in Figure 5.15.

5.6.3 Image Preprocessing

Different illumination conditions result in different token colors on the game board. The marker is, due to its vast contrast (black-white), more invariant to illumination. This contrast is achievable through the binary entity of the marker. Of course, the marker is neither visible in total darkness nor in direct sunlight, but under normal conditions, for example daylight.

The colored tokens are not that invariant to light. During the day, the color of the tokens differs. This challenge is solved by an initial calibration at game start, which enables the robot to recognize the currently available colors. This also offers the possibility to use every desired color, however they should be different from another. The color detection is achieved in the HSV color space, because it is more invariant to light conditions, due to having an extra channel to adjust the illumination level. In the RGB color space a slightly different illumination change already results in another token color and therefore it is more difficult to detect a specific color than in the HSV color space. As mentioned in Section 3, a color in the HSV color space can be found within simple thresholds. In the RGB color space a color is located in a conical area, which is more difficult to model. Although the NaoQi is able to already transmit the images in the HSV color space, the native image format is YUV422. In both cases the camera images have to be converted. I experienced that the conversion to HSV is much slower than a conversion to RGB. Hence the image is requested in the RGB color space. Moreover both color spaces are used by the vision. The image in the HSV color space is used to segment the colors, while the image in the RGB color space is used to visualize the images.

Better results with a fixed tolerance than with variable approaches were achieved, because although the lighting conditions change, they never change completely. With variable tolerances, we sometimes had too far extended or too restricted thresholds. In these two cases, either environment was recognized to be tokens or tokens were not recognized at all. As we implemented an approach for variable thresholds, we noticed that an approach with fixed thresholds is more robust and therefore more reliable for this task.

We wanted to remove the illumination effect and enhance the colors. It was expected that a better contrast would support the segmentation step of the tokens. Therefore a histogram equalization for colored images was implemented. The normalization was successful, the contrast increased but also the level of detail. As a side effect shiny regions were now classified as white and tokens with strong edges or cracks consisted of multiple colors. Independent to the illumination, we always had an image with high-contrast, but also modified color values. This results was not satisfying as we wanted to enhance the colors and decrease the level of detail. In the next step we tested the normalization in combination with a Gaussian blur. The Gaussian blur reduced the level of detail before and after the normalization, but the image still contained volatile color values. So we disabled the normalization and just used the Gaussian blur, which worked much better.

A color search in the blurred image in the HSV color space is made to find the colored tokens. The desired color is taken from the calibration tokens at the beginning of the game. Therefore the median color value over a small pixel area in a circle of 5x5 pixels is taken for the given image position. This image position corresponds to an ideal position that is occupied by a token from player one. The color on that position is assigned to player one. As a matter of fact,two players are required for this game. The robot has to move its head and the calibration process has to be started again, assigning the found color on the second position to player two. From now on, both players are regarded to be initialized and their color may be tracked. Using this procedure to select the color offers the possibility to use any desired color. Two different color tokens are required at startup and the tokens may be tracked the whole game. Additionally a third color, a blue color, is also tracked. The color for calibration is mounted on the robot's shoulder and is initialized the same way. It is used to track the blue stripe on the side of the board. So it is not recommended to use blue tokens, even if it would be possible. More information regarding the game token color can be found in Section 5.8.

Good lighting conditions improve the marker detection, because it can be tracked more exactly and therefore the token positions are fitting better to their real positions. Consistent lighting conditions are well-suited for the token detection, because the color values only change marginally. After the Gaussian blur with a kernel size of eleven and the conversion of the RGB image to the HSV color space has been

applied, the color detection starts. There is an optimized OpenCV [32] function that does exactly what is required in this step. The function ”inRange” returns a binary image for a given color image with a lower and an upper threshold. The threshold is equivalent to the previously initialized color, minus or plus a fixed tolerance. All color values in this tolerance return a value unequal to zero (non-black). This binary one-dimensional image may also contain ”false” detected pixels. In this case ”false” means that pixels do not belong to a token of the requested color.

In the next step, we try to filter single non-black pixels that are created by image noise. On each position in the binary image the number of non-black neighbors are counted in a 5x5 neighborhood excluding the center dot. If there are at least 9 non-black pixels, the center pixel remains white. Otherwise it is classified as noise and assigned to a black pixel. This is important, because the next procedure is a morphological close, which would enhance small false-detected structures, but also completes not fully recognized tokens.

Afterwards we receive a binary mask that contains the tokens as blobs. This mask often contains some minor artifacts, like single white pixels or a tattered border. Therefore a morphological ”close” is applied to the binary image, which removes most of the artifacts. Now the image preprocessing is complete. The result of the image preprocessing step can be seen in Figure 5.17. In this figure both blob images and the currently hardly visible blue stripe are drawn inside the current camera image for better visualization and verification. The next step it is required to identify which blob belongs to which strategic position. This is calculated in the vision class that classifies all tokens and is discussed in Section 5.6.6.

Figure 5.17: Final color detection, colored and automatically drawn in a camera image

In addition to the blob detection, this class offers a service to check, if there is a token in the hand. For this task four information sources are required:

- The player ID of the token to recognize (own or opponent token).
- The hand status (open or closed hand).
- The used arm (left or right).
- The current image.

The first three sources are received together with the service request. The current image is globally available for read access in this class. At first the blob image of the requested player is generated. This blob image is compared to one of four masks, where one mask corresponds to the arm and hand status. The four masks are manually premanufactured for the specific arm pose and hand status. If a certain amount of common pixels are found in both images, it is most likely that a token is in the hand. The masks in the form of Nao’s fingers are important, because it may be possible that the board with the game tokens is in the background. Without the masks the robot could recognize a token in the hand, even though there is none. This results of tokens on the board that are visible in the background between its fingers. The background region behind the three fingers is larger than the region of the fingers, which would also lead to colored pixels of the desired player. A detection procedure is visualized in Figure 5.18.

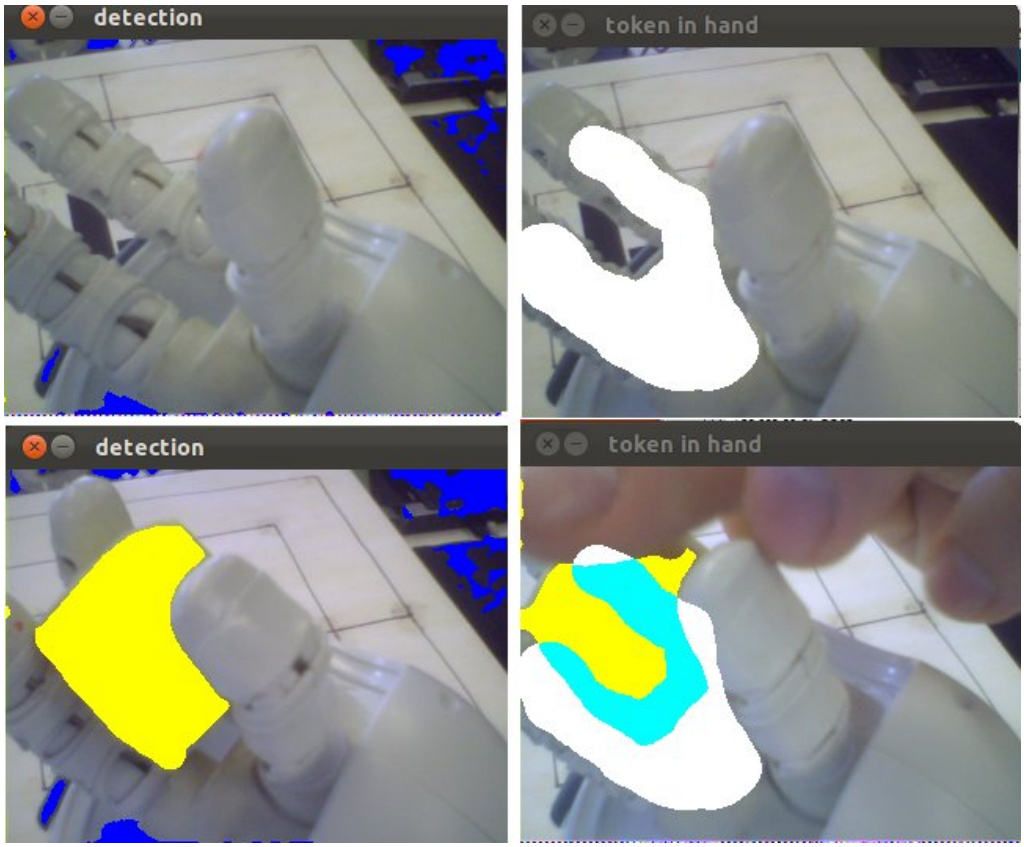


Figure 5.18: Images for the detection for tokens in the hands. The top left image shows the open right hand. Top right shows the same with an overlain mask, which is used to check, if there is a token. Bottom left shows the hand with a token. Bottom right shows the yellow token that has been detected in the teal area. The teal area shows the overlap of the mask and the token. The yellow area is not rated, because it could also be background information.

Another function is responsible for the blue stripe detection. This function is called, if the robot is searching for the marker. In case the marker is not found, blue stripe information is available. This function detects the nearest blue color in front of the robot, corresponding to the lowest row of a blue point in the image. Then the distance to the blue object is estimated, using the row in the camera image and the current camera height and roll. The estimation of the blue stripe distance is explained in Section 5.6.4. If the marker detection failed, the robot walks the estimated distance, minus its size, towards the blue stripe. Now that the robot should be closer to the board, it should be possible for the robot to detect the marker again. The blue stripe on the border of the board can be seen in Figure 1.2

5.6.4 Estimation of Distances and 3D Points

A grasp procedure requires a real token position, in contrast to a place that only requires the ideal token position. In the first approach the real position was estimated, using the camera pose and the center of the target blob in the image plane. The blob center has been projected to a three-dimensional position on the game board, considering the aperture angles of the camera. This approach was not very accurate and became obsolete, since the blob image is rectified. Due to the rectification that is required for classification, the estimation of the real position is a simple task. The only challenge is to acquire the focus point, which is used to determine the four points to rectify the image. The approach to find the focus point is similar to the first approach but only with the center image coordinates. This facilitates the calculation, because the

camera angle in the transformation stream can be used for the focus point calculation without any offset.

Focus Point

The computation of the focus point requires the height Δ_height_{cam} of the camera above the game board and the camera angle α to the horizontal. Both values can be requested from a transformation of the bottom camera in a foot link. This yields the height $height_{cam\_foot}$ of the camera to the ground and a camera angle α_{foot} that is rotated by 90°, because the z-axis of the foot corresponds to the y-axis of the camera and therefore the angle has to be rotated as shown in Equation 5.4. For better understanding a draft of the focus point estimation can be found in Figure 5.19.

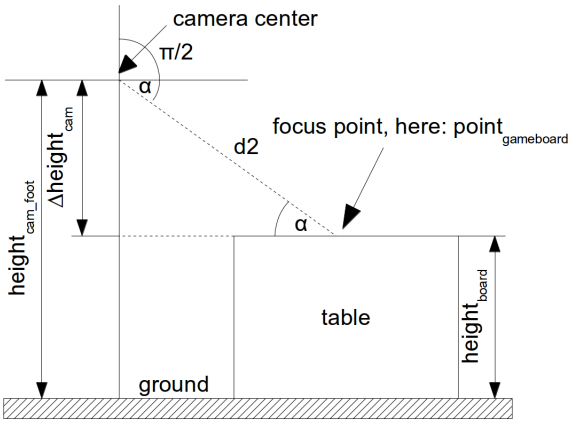


Figure 5.19: Draft of the focus point on the board

$$\Delta\_height_{cam} = height_{cam\_foot} - height_{board} \tag{5.3}$$

$$\alpha = fabs(\alpha_{foot}) - \frac{\pi}{2} \tag{5.4}$$

Using the sine with the calculated height Δ_height_{cam} and angle α, yields the distance d from the camera center to the focus point on the board (see Equation 5.5). The line from the camera center to the focus point is always the z-axis in the camera system. The focus point in the camera system is, per definition, zero for x and y. Therefore this point is transformed into the game board system using the ROS transformation (see Equation 5.6).

$$d1 = \frac{\Delta\_height_{cam}}{sin(\alpha)} \tag{5.5}$$

$$point_{gameboard} = transform\_cam\_to\_game\_board(0,0,d) \tag{5.6}$$

This results in the estimated point $point_{gameboard}$ on the board. The height of this point deviates from the expected one, which requires a correction step. The difference $point.z_{gameboard}$ in the expected height has to be divided by the sine of α, as shown in Equation 5.7, and added to the previously calculated distance $d1$ and transformed again to achieve a better real position for the focus point, as shown in Equation 5.8.

$$d2 = d1 + \frac{point_{gameboard}.z}{sin(\alpha)} \tag{5.7}$$

$$point_{gameboard} = transform\_cam\_to\_game\_board(0,0,d) \tag{5.8}$$

Now the refined point fits to the real focus point and can be used for the rectification procedure.

Real Token Positions

The computation of the real token position is a simple task after the image has been rectified and mapped to the combined mask. At first, the center pixel coordinates of the token that has to be grasped, is calculated in the rectified blob image. Afterwards the half mask size is subtracted from the image coordinates and they are divided by the mask-to-game-board-factor $\left(700.315\frac{pixels}{meter}\right)$(see Section 5.6.6) to convert pixels into meters in the game board system.

Blue Stripe Distance

The blue stripe has been mounted on the board for a course localization support. The robot tries to estimate the minimal distance to the board, using the lowest row in the image that contains blue values and the camera roll angle. A lower row in the image corresponds to closer objects, while a higher row refers to more distant objects. In contrast to the focus point estimation, the blue stripe distance estimation considers the row in the camera image. The distance should be calculated in torso coordinates for direct usage of the walk planner.
For better understanding there is a draft of the calculation shown in Figure 5.20.

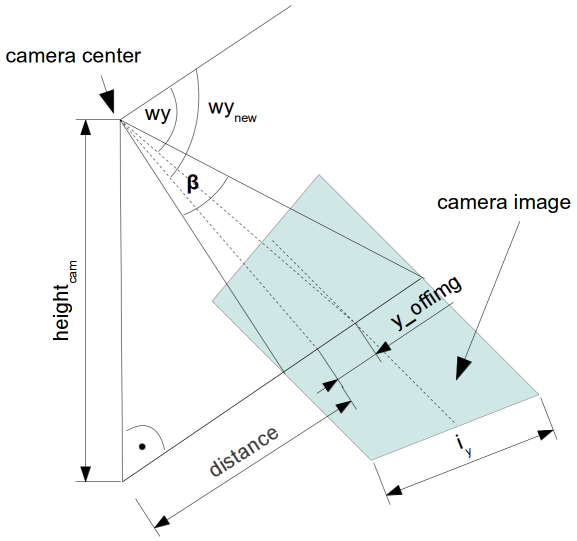


Figure 5.20: Draft of the blue stripe distance calculation. The camera image that is projected on the board plane can be seen with the y-offset, the new angle and the resulting horizontal torso distance.

The origin of the camera is in the center of the image, while the pixel coordinates of the row are counted from the bottom image border. Therefore the row coordinate has to be shifted by half of the image size (see Equation 5.9) to receive the y offset y_off_{img} from the origin. Additionally the $\frac{degrees}{pixel_y}$ (Equation 5.10), which are calculated by the vertical aperture angle β and the image height i_y, are required to compute the new view angle of the blue stripe. The new angle wy_{new} is computed, by the current camera angle wy plus the offset that is converted to an angle, as shown in Equation 5.11. The camera height is computed the same way, which is shown in Equation 5.3. The minimal horizontal distance to the blue stripe from the torso is estimated, using the new height, the tangent of the new camera angle and the offset from the camera in front of the torso x_off_{cam}.

$$y\_off_{img} = row - \frac{i_y}{2} \tag{5.9}$$

$$\frac{degrees}{pixel_y} = \frac{\beta}{i_y} \tag{5.10}$$

$$wy_{new} = wy + \frac{y\_off_{img} \cdot \frac{degrees}{pixel_y} \cdot \pi}{180} \tag{5.11}$$

$$distance = \frac{height_{cam}}{\tan(wy_{new})} + y\_off_{cam} \tag{5.12}$$

5.6.5 Projection of the 3D tokens into the Image Plane

This projection maps a visible point of the three-dimensional space into a two-dimensional point on the image plane. We experimented with different approaches. The first intuitive approach did not consider the intrinsic parameters directly but worked very well after a short calibration. In all cases the points are available in the camera coordinate system, therefore the extrinsic parameters are negligible.

The first approach is based on the following considerations. Figure 5.21 shows the camera with its image plane and a three-dimensional object in a perpendicular view to the x-axis and z-axis of the camera.

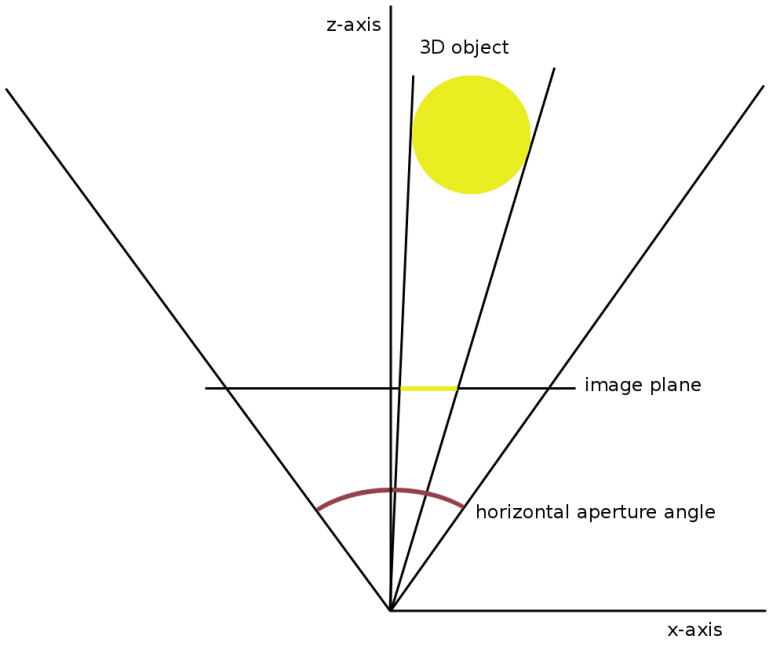


Figure 5.21: Projection of three-dimensional point of the scene into the image plane of the camera

For a three-dimensional point, given in the camera coordinate system, the z-axis is the perpendicular ”distance”, while the x-axis and y-axis are the offset from the camera origin. The camera origin should be the middle of the image plane without considering the principal point offset. Assuming we have an aperture angle of 90° and a x and z value of 1 m. This means that the object is 1 m in front of and 1 m right to the camera center. A division of the offset to the right x by z results 1. This result would correspond to the most right possible position on the image plane. Now the resolution of the image plane has to be considered. Owing to the fact that the x and y start from the center of the image, the half image width has to be multiplied with the resulting value of $\frac{x}{y}$. An image width of 320 pixels concludes to a range from -160

to +160 pixel. Furthermore the origin of the image plane points should be in the bottom left corner. So another offset of the half image width has to be added to the result. For now our formula with the example looks like Equation 5.13.

$$x_{image} = \frac{\frac{width_{image}}{2} \cdot x_{cam}}{z_{cam}} + \frac{width_{image}}{2} \tag{5.13}$$

$$x_{image} = \frac{160 \cdot 1}{1} + 160 \tag{5.14}$$

$$x_{image} = 320 \tag{5.15}$$

For real cameras, the aperture angle is not 90° and the image plane is not square shaped. In the case of Nao's cameras, we measured the foremost right, left, top and bottom pixel. For x an absolute maximum values of 0.4, and 0.3 for y were received. Thus a distance of one meter on the optical axis from the image plane to the object, the object could be 0.4 m to the left or right and 0.3 m to the top or bottom that the camera can barely see it. We normalize our z value to receive 1.0 on the greatest extend of x and y. The resulting formula in the real world case with the horizontal x value is shown in Equation 5.16. The same formula can the applied for y with a factor for the aperture angle of 0.3 instead of 0.4.

$$x_{image} = \frac{\frac{width_{image}}{2} \cdot x_{cam}}{z_{cam} \cdot x_{aperture}} + \frac{width_{image}}{2} \tag{5.16}$$

$$x_{image} = \frac{160 \cdot 0.4}{1 \cdot 0.4} + 160 \tag{5.17}$$

$$x_{image} = 320 \tag{5.18}$$

This naive approach offers points in the image plane that approximate the three-dimensional points nearly as well as using a projection matrix.

The second approach uses the projection matrix P that projects a three-dimensional point into the image plane in homogeneous coordinates. This approach is similar to the first one, but also respects the intrinsic camera parameters K. K is a 3x3 identity-matrix that contains the focal length f and the principal point offset t of the camera model as shown in Equation 5.19. Rarely cameras also have a skew, which means that the chip plane is not perpendicular to the optical axis of the camera. This could be modeled with the second intrinsic parameter of the first row of K, but equals to zero in most cases.

$$K = \begin{bmatrix} f_x & 0 & t_x \\ 0 & f_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.19}$$

The extrinsic parameters are a rotation matrix R and translation vector $\vec{t}$ to define the pose of the camera in the world. The three-dimensional points already were transformed into the camera coordinate system and the extrinsic parameters just need to be set to the identity matrix, so that the Equation 5.20 for the projection can be simplified.

The three-dimensional point itself is a vector I with three elements. As the result we receive a three-dimensional vector P in homogeneous coordinates. For dehomogenization x and y have to be divided by z. The simplified formula is shown in 5.21.

$$P = K \cdot R \cdot \left[I \cdot \vec{t} \right] \tag{5.20}$$

$$P = K \cdot I \tag{5.21}$$

The result with this approach did not improve. This could correspond to uncertainties of the principal point offset or the focal length, which were calculated with the Caltech toolbox [9]. This toolbox is based on the approach of Zhang [46], who proposed a flexible and simple way to calibrate common cameras. The calibration target that has been used is shown in Figure 5.22, where 30 pictures were taken of different poses of the target.

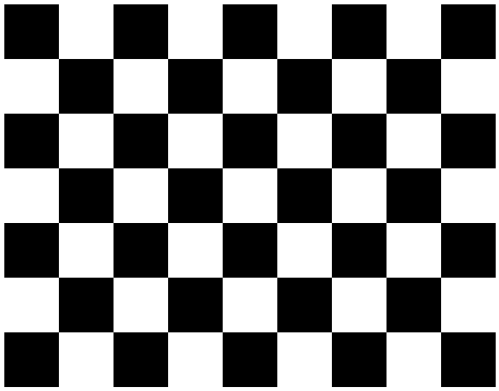


Figure 5.22: Calibration target to obtain the intrinsic camera parameters

5.6.6 Blob Classification

The classification is a challenging task, due to occlusions of tokens that appear under certain angles of view and transformation uncertainties that also depend on the accuracy of the marker detection. The marker accuracy depends on the illumination and the angle of view. Hence there were several methods implemented to reduce uncertainties and handle logical conflicts to make the classification step more robust.
The blob classification requires the binary blob image to classify each blob. Furthermore there is an additional helper library ”BwLabel”, discussed in Section 5.6.7 and class ”NaoHelper”, explained in Section 5.5, that are designed to simplify the further processing of the blobs. Before the binary images are labeled, they are rectified and mapped to another fixed size matrix. In an earlier version this step was not required, because the blobs were directly evaluated by their overlap with the dilated ideal position. This is an efficient method and valid solution but performs badly, if a token does not lie next to an ideal position. Therefore a solution was required that rectifies the blob images and compares them to manually prepared masks. The combined image of all comparison masks has already been shown in Figure 1.4 and represents a perfectly rectified game board.

Due to the fact that rectification is a challenging task, it is described in detail in Section 5.6.6. After the rectification procedure of the blob image, all blobs have to be labeled, because at this very moment all blobs together are segmented from the rest only.

The ”BwLabel”-library offers a function called ”getBlobPixelList”, which takes the blob image that is passed by the Classification-class and returns a vector with a vector of points. Each of these point-containing vectors contains all pixel coordinates of one blob. Furthermore ”BwLabel” offers the function ”getLabeledMatrix” that returns a matrix, where each blob is assigned to a value (see Section 5.6.7).

Now that each blob pixel size is known and the image is rectified, we could directly make an assumption about the probability of a blob. In the earlier version, the blob probability was estimated with the number of pixels of the blob and its distance that was estimated with the camera pose and the position in the image. This led to challenges in the estimation of real positions. Although they could be estimated, it was only possible to do so next to an ideal position, which sometimes led to an insufficient accuracy.

The current approach rectifies the blob image. On the assumption of an ideal rectification, all blobs should have more or less the same size with a specific tolerance for distortion. Additionally the estimation of the real position is much easier, because the blob position in the rectified image can directly be converted into a game board position, with an offset and a factor for each rectified position.

There is a loop that iterates through all 24 positions checking every blob for every position. The decision, if a blob lies on a position, is made by the amount of pixels that result by a ”binary AND” of two matrices. The first matrix contains the mask of one strategic position and the second mask contains the rectified

blob image with the same orientation as the mask. If there is a minimum amount of common pixels, there definitely is a blob. In this case the probability for that player's color is increased, the timestamp for that position is set and the three-dimensional position is estimated. This is the common procedure for one image. Due to the fact that it is not possible to see the whole game board at once (see Figure 5.23), there are multiple images needed to acquire the whole game state. There has to be a border condition that ignores halved tokens, as the red token on the right side of the middle camera image of the last mentioned figure. Only complete tokens are allowed to be evaluated and filled in the "board information". This is essential for the world model to know, which positions have already been updated and which positions are still unknown. If all positions are observed, the game state is up to date, until a token is placed or removed.

If everything works as expected, the blob classification returns a filled "board information" message with a status code "0". A code lower than "0" shows that an error occurred and the information is invalid or compromised. It contains information for every strategic position, like the probability for each player, the estimated three-dimensional position and a time stamp of the last update. Furthermore the "BoardInformation" contains information about where detection conflicts happened. The two different conflicts that may occur are explained in the two next sections.

Multiple Blobs

Multiple blobs are defined as blobs of the same color that melted together to one blob in the binary blob image as shown in Figure 5.23. On the one hand this results partly on the morphological close effect that is applied to the blob detection. This effect reduces disturbing objects in the image but also leads to melted blobs. On the other hand, adjacent blobs already appear like one blob under perspective observation. Usually a melted blob has a higher pixel size than a single blob. In an earlier version, multi blobs led to challenging tasks in the detection of the real position but in the current approach the position of a blob is only computed on the local mask at each strategic position. Although multi-blobs are still a challenge, this strategy results in a far more accurate position of each blob.

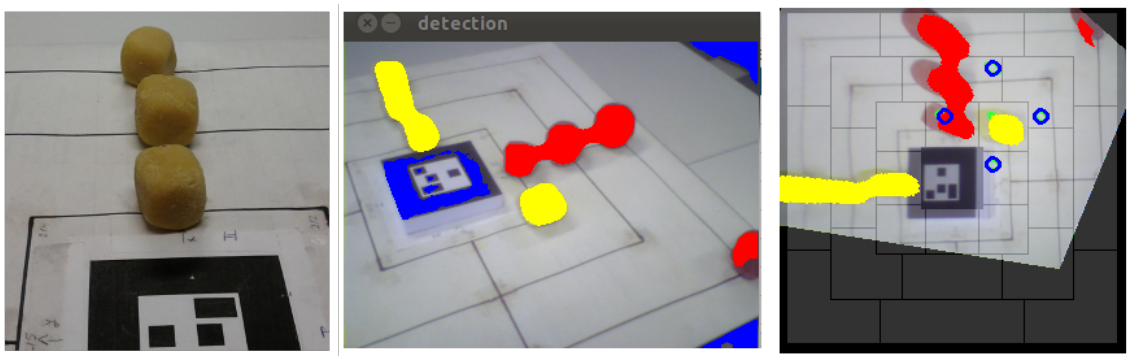


Figure 5.23: Example of multiple tokens detected as one blob. The image on the left shows tokens from an external camera at the height of the Nao. Note that there is no visible white space between the tokens of the same color, although they are placed perfectly. The middle image is the camera view and token detection of the robot. The three tokens of the same color that lie in a line are detected as one big blob. The image on the right is the rectified camera image combined with all masks.

Conflict Tokens

Conflict token are generated, if there are apparently more than one token in one mask. An example for a conflict token can be seen in Figure 5.24. Conflict tokens have been implemented as repair strategy for place actions. If a token cannot be recognized after it has been placed, the game board is updated. The conflict value of the "board information" is set to the index, where two tokens are recognized in one mask.

Additionally the real position of the second token is saved. Now the robot knows, were its lost token is gone. If the conflict value is not set, the difference of the updated and the last game state is extracted. If there is a difference, it is obvious that the lost token rolled to this previously unoccupied area. Afterwards the robot can correct the misplacement by grasping and replacing the token. In case no conflict token or difference is found on a misplaced token, the token may have fallen of the table and human help is required.

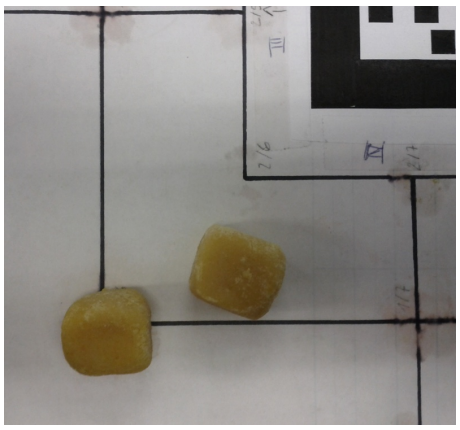


Figure 5.24: Example of two separate detected blobs in one mask

Rectification

The rectification procedure should apply a projective transformation to the current camera image. This is accomplished by a mapping from the first into the second image. The mapping is also called homogeneous matrix H. The following paragraphs for the calculation of the homogeneous matrix H are based on the book of Hartley and Zisserman [21, p. 33-35]. As a matter of fact, an image transformation is two-dimensional and calculated by a 3×3 matrix, as shown in Equation 5.22. The image coordinates (x, x') are homogeneous.

$$\begin{pmatrix} x_1' \\ x_2' \\ x_3' \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{5.22}$$

$$x' = Hx \tag{5.23}$$

The matrix H can be multiplied with a non-zero factor and will also result in the same transformation, because only the ratio of the matrix elements is important for the transformation. The nine elements yield eight independent ratios that lead to eight degrees of freedom. The eight degrees of freedom consist of:

- $2\times$ translation
- $2\times$ rotation
- $2\times$ scale
- $2\times$ line at infinity

Invariant in the projective transformation are the cross-ratios, as points will always have the same order before and after the projection. This fact results on the central projection through one common point. Considering multiple lines, as one is visible in Figure 5.25, from the common projection point o, to a point in the first image will always result in the same order of points in the second image.

Usually an image is distorted under perspective view as shown in Figure 5.27a, one can see the original distorted camera image. In the top region of the image one can see that the lines are not rectangular,

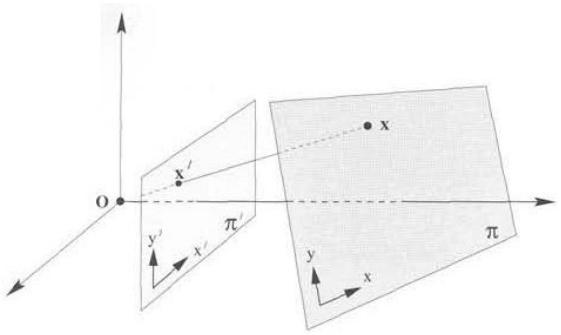


Figure 5.25: The plane mapping of a projection [21, p. 34]. It shows that based on the common projection point o, the cross-ration is preserved.

although in the scene they are. The lines appear to converge at a finite point. Applying a projective transformation H to the image will remove the distortion, showing the image in its correct geometric shape. The resulting rectified image after the transformation is shown in Figure 5.27b, where all angles of the game board lines are rectangular. The computation of the transformation matrix requires corresponding inhomogeneous points of the images, where x is the point in the first image and $x^{'}$ is the point in the second image.

$$x^{'} = \frac{x_1^{'}}{x_3^{'}} = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \quad y^{'} = \frac{x_2^{'}}{x_3^{'}} = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{5.24}$$

Since the points in both equations of 5.24 are known, they generate two equations per point correspondence to calculate the transformation H. Equation 5.25 and 5.26 show the two equations for one point.

$$x^{'}(h_{31}x + h_{32}y + h_{33}) = h_{11}x + h_{12}y + h_{13} \tag{5.25}$$

$$y^{'}(h_{31}x + h_{32}y + h_{33}) = h_{21}x + h_{22}y + h_{23} \tag{5.26}$$

Due to eight degrees of freedom and two equations per point correspondence, four corresponding points are required to solve H. The precondition for each of those points is to be currently visible and collinearity of more than two points is strictly prohibited. As an image has to be rectified, four points in a rectangular turned out to be the best choice. The scale factor of H cannot be retrieved, but as mentioned, it is useless for the transformation.

The resulting rectified image has always the same rotation in relation to the marker, regardless of the position of the robot due to the point selection. A rectification from another board side was already shown in Figure 5.23.

As the OpenCV library [32] already has functions to compute the transformation, corresponding points need to be found. For this purpose we created four points in the plane of the game board that form a square. These four points are selected all around the current focus point of the camera to ensure that all four points are visible. This also implicitly defines that it is never possible for three points to lie on a line. Additionally the distance between all points is proportionally increased with the distance from the camera to the game board. The distance is measured from the center of the camera along their z-axis to the intersection of the board. This is required, because with increased distance, the rectangular shape appears smaller in the camera image and the resulting computation of the transformation would be more inaccurate. A good value for the distance from the focus point to each of the four points turned out to be $\frac{focus\_point\_distance}{7}$. Once these four points are set, they have to be transformed in the coordinate system "CameraBottom_frame" (see transformations in Section 5.3.4). The two-dimensional positions of the first four points are now retrieved by projecting the three-dimensional camera points to the image plane (see Section 5.6.5).

In the next step, the paired four points have to be computed. Again we take the same four points in the game board system. These points have to be converted from meters into pixels. The conversion factor is calculated by dividing the side lengths of the outer squares, as shown in Equation 5.27 and visualized in Figure 5.26. The side length of the outer square of the mask is calculated by subtracting two times the border b to the first square from the complete side length of the mask m. The side length of the outer square of the real game board is calculated by the sidelength of inner square $d1$, plus four times the distance between two squares $d2$.

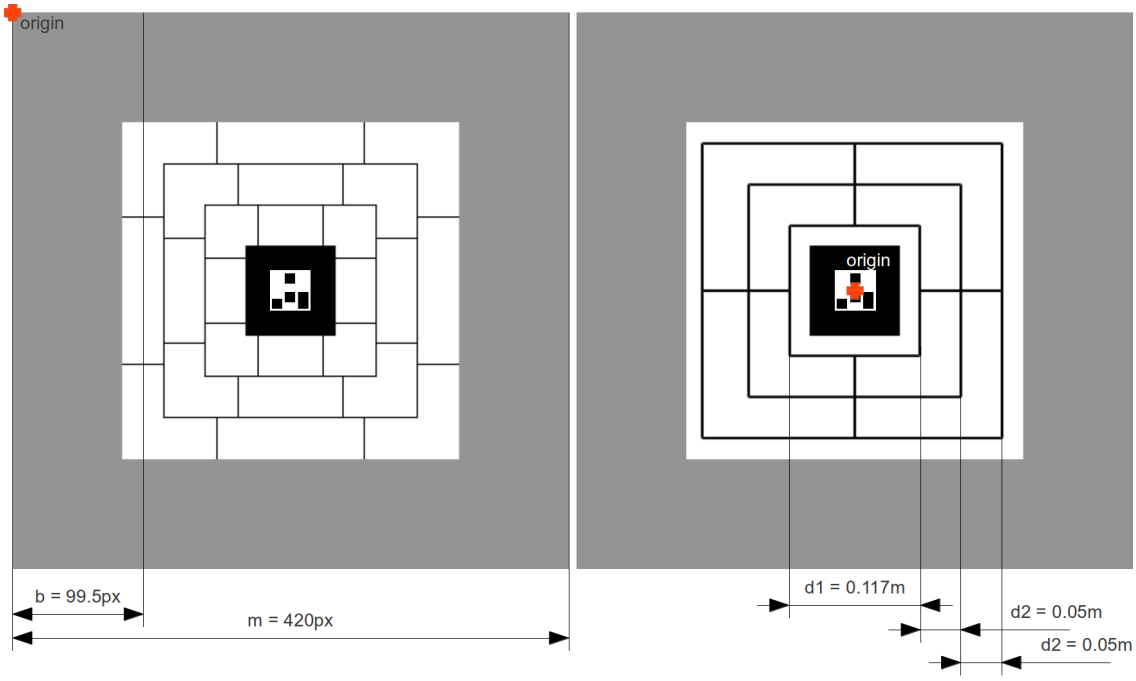


Figure 5.26: Sidelengths of the abstracted combined masks and the real game board to calculate the conversion factor. Additionally the origins of both systems are shown with red cross.

$$\frac{pixels}{meter} = \frac{m-2\cdot b}{d1+4\cdot d2} \tag{5.27}$$

$$\frac{pixels}{meter} = \frac{420-2\cdot 99.5}{0.117+4\cdot 0.05} \tag{5.28}$$

$$\frac{pixels}{meter} = 700.315 \tag{5.29}$$

After the conversion to pixels, the point needs to be shifted by an offset value that corresponds the offset from the origin of the game board system to the origin of the mask. The origin of the mask is located on the top left corner while the origin of the game board is located in the center of the board. So the offset value is defined by the half of the complete side length of the mask m as shown in Equation 5.30.

$$offset = \frac{m}{2} \tag{5.30}$$

Algorithm 2 shows the summarized calculation of all eight points.

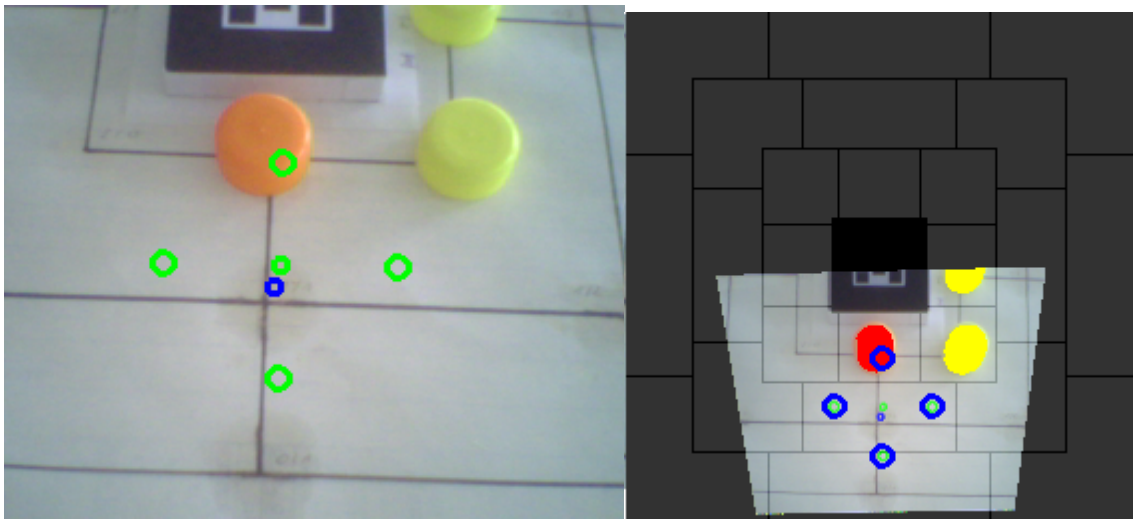
Now, that the two-dimensional positions of all eight points are known, the homography may be estimated, using the OpenCV function ”getPerspectiveTransform” [32]. Afterwards the current camera image

Algorithm 2: compute eight points for rectification

input : fp ... three-dimensional focus of the camera in game board system,
$distance$ ... distance between camera center and focus point on the board
output: $points_{image}$ ... four points in the image
$points_{mask}$ ... four points in the mask

$size\_square = distance \cdot 7$
$point[0]_{board} = fp.x - size\_square$
$point[1]_{board} = fp.y + size\_square$
$point[2]_{board} = fp.y - size\_square$
$point[3]_{board} = fp.x + size\_square$
$pxm = 700.315$
$point_{mask}.x = point_{board} * pxm + offset$
$point_{mask}.y = point_{board} * pxm - offset$
$point_{camera} = transfromPointsFromGameBoardToCamera(point_{board})$
$point_{image} = project3DPointsIntoImage(point_{camera})$

may be rectified by the function ”warpPerpective”. The order of the points is very important due to the invariance of cross-ratios. Otherwise the computation of the correct transformation will fail. Although it will be rectified, the orientation will be wrong. The orientation has to always be the same in relation to the marker. Otherwise the tokens cannot be classified correctly. The order of these points are always known, as they are chosen depending on the game board coordinate system. An introduction for affine transformations using OpenCV can be found in [10]. The original and the rectified blob image combined with all masks are shown in Figure 5.27. The uncut rectified image has a black border, as shown in Figure 1.4. This is used to cope with border conditions, because it is not guaranteed that one of the four points to rectify the image lies inside the game board area.



(a) Original RGB camera image (b) Rectified image

Figure 5.27: Both images show four points that are all all round the current focus point in green. These four points are the point correspondence that were used to compute the projective transformation H. The blue dot in the middle is the center of the image. In Figure 5.27b, the rectified camera image is overlays with the masks. Additionally the tokens are colored, depending on their detection and classification.

5.6.7 BW Label

The BW Label class is used to simplify the blob image handling. There are two public methods to process a binary blob image. Both take the same input parameters and are labeling the blobs, but they differ in their kind of return parameters. The first method "getLabeledMatrix" returns a matrix of same size as the input image and is filled with chars. The difference to the binary input image is that all pixels of one blob now have the same value. From top left to bottom right the blobs are labeled in ascending order from 1-255. This also limits the maximum blobs to 255, while the background receives the value 0. Due to morphological operations on the player blob images, it is not possible to exceed the maximum blob value of 255. If the blobs would be placed optimal equidistant and not overlapping, the maximum number of blobs num_blobs_{max} after a morphological close with the size of 20x20 on an image of the size of 320x240 would be 196, as calculated in Equation 5.32.

$$num\_blobs_{max} = \frac{320}{20} \cdot \frac{240}{20} \tag{5.31}$$

$$num\_blobs_{max} = 196 \tag{5.32}$$

The second method "getBlobPixelList" returns a vector with vectors of points. Each of the vectors that contains points corresponds to one blob. Each point of the vector contains the image coordinates of the pixel. On the one hand it is easier to count the number of pixels of a specific blob or address its pixels. On the other hand, the first method is easier to visualize.

Both methods are using the same algorithm. There is a loop through all pixels in the binary blob image. If a non black pixel is found that has not been processed already, it is added to the labeled matrix and to a vector. Then all non-black pixels in the 8-neighborhood are examined. If they have not been assigned yet, they are added to the labeled matrix and to the vector. This procedure is repeated until there are no non-black pixels that have not been assigned yet. Only then, the loop continues and repeats this process until it reaches the last pixel and returns.

5.6.8 Visual Odometry

The visual odometry was implemented to support the localization during walk procedures. Sometimes a walk failed, because the rotation about the vertical robot axis was off by about 10°. A forward motion of about 0.5 m then resulted in an offset of about 0.09 m, which sometimes led to a collision of the robot and the table. This rotation error is calculated by the visual odometry to correct the robot rotation.

The visual odometry is now active during a walk procedure. Before a large rotation ($> 45°$) is made, the robot rotates its head to the target angle and takes a picture. Afterwards the head is rotated back and the torso is rotated to the target angle. The robot takes a picture again. Assuming the head rotation is more precise than the torso rotation, the robot has now taken two similar pictures. This is possible due to a nearly pure camera rotation. Both images should only be shifted slightly along the horizontal. However, they are not similar, if there are many changes between the capture of the two images.

Both images are analyzed with the help of the OpenCV library [32] . First of all a keypoint detector is applied to find interesting points in each image. Raw interest points are useless, so descriptors of this points have to be made. The SURF descriptor was chosen, because it combined the robustness and speed, which is required to solve this task. After the descriptors are calculated for both images, the next task was to find matches between both images. A "radiusMatch" function was used that finds matches for descriptors in a given distance.
The matches have to be evaluated now, They are shown in Figure 5.28.

A perfect robot rotation should result in the same image and keypoints. In real world applications there always is a deviation. We wanted to retrieve the offset of the matches in pixels in the horizontal direction.

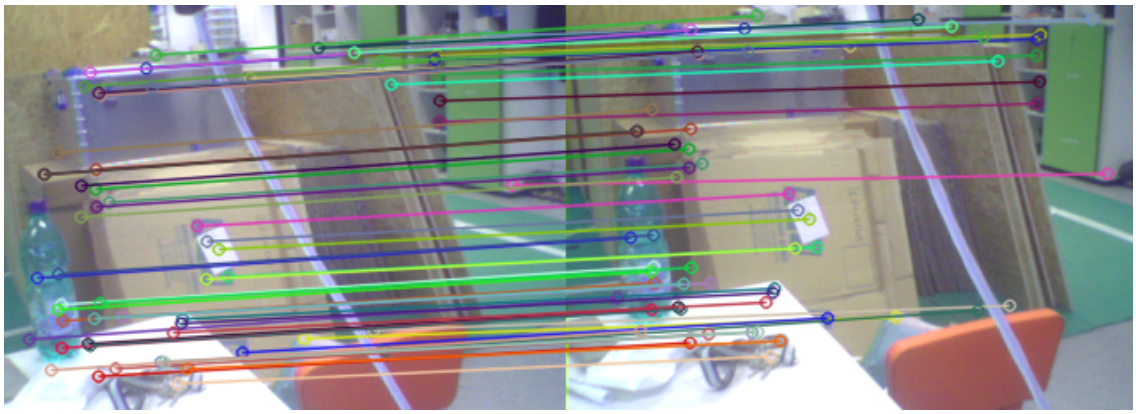


Figure 5.28: Sample of two images after rotation with visualized matches.

The offset of each match is computed, sorted and the median value is taken. If there are enough matches (by definition > 9), this pixel offset has to be converted to an angle, which the robot can correct. If there are too little matches, the visual odometry fails. This results from too little keypoints, which most of the time is the result of insufficient light.

The relation from the offset of the pixels in the image pair to angles is not affected by any distance of the robot to its environment. A simple rotation of the camera around one fixed point can never estimate a distance. It does not matter, if a detected point is 0.5 m or 5 m away, because only the position in the image is important. Any point in the three-dimensional space, which is visible in the camera, is projected into the image plane and results in a pixel without any depth information.

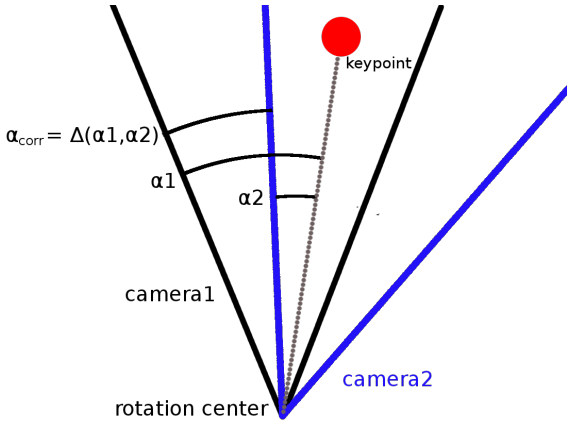


Figure 5.29: Illustrates the difference angle α_{corr} computed of two similar camera images.

Figure 5.29 illustrates an observation of the same keypoint that is visible in two similar images. Assuming a camera rotation in a common point, yields a rotation angle α to one keypoint in both images. The difference of both angles corresponds proportionally to the offset in pixels in the image. With sufficient matches, the degrees of the correction angle α_{corr} can be calculated with the horizontal median offset $offset_{pixel}$, the image width i_x and horizontal aperture angle β by:

$$\alpha_{corr} = offset_{pixel} \cdot \frac{i_x}{\beta} \tag{5.33}$$

$$\alpha_{corr} = offset_{pixel} \cdot \frac{320}{44.865} \tag{5.34}$$

$$\alpha_{corr} = offset_{pixel} \cdot 7.1325 \tag{5.35}$$

5.7 Motion Package

The motion package contains the ”NaoMovement“ node, which executes the packed commands received from the planner package by sending it in the correct form to the NaoQi. Each command is checked for valid angle values. The tasks of the motion package are summarized in Figure 5.30.

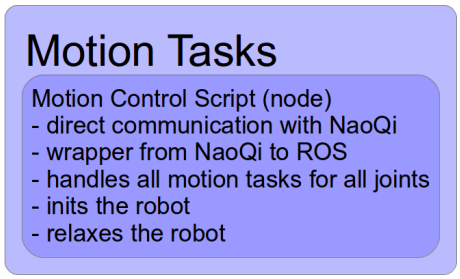


Figure 5.30: Motion tasks

Additionally the motion package is responsible to initialize all joints of the robot. Therefore it raises and lowers the torso and moves all arms and the head to an init position. The init procedure moves all of the joints to update the sensor information of the joints. Without the init, the robot sometimes performs strange trajectories, while executing a path. We think this results of uninitialized or invalid joint positions at startup.

The relax function also is implemented in the motion. If it is called by the planner, the robot sits down and removes its stiffness. This relax is called after each turn to switch off the motors to preserve their functionality. Otherwise the motors overheat and the robot is losing stiffness. This could result in breakdown while walking.

5.8 Game Tokens

Due to the amount of game tokens, it was more economical to produce them on our own. This way, it was possible to choose the preferred colors that could be easier detected by the vision. In the color evaluation, yellow color could be detected best. Followed by purple, red, blue and green. Although red is very different to yellow, it is more challenging to detect red in the HSV color-space, because red is at the start and end *hue* value of the circle of the color cone. This means red can have the *hue* value of 0 or 2π, as can be seen in Figure 5.31.

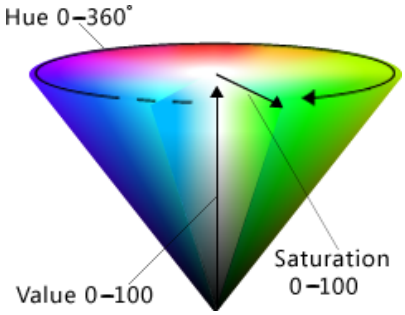
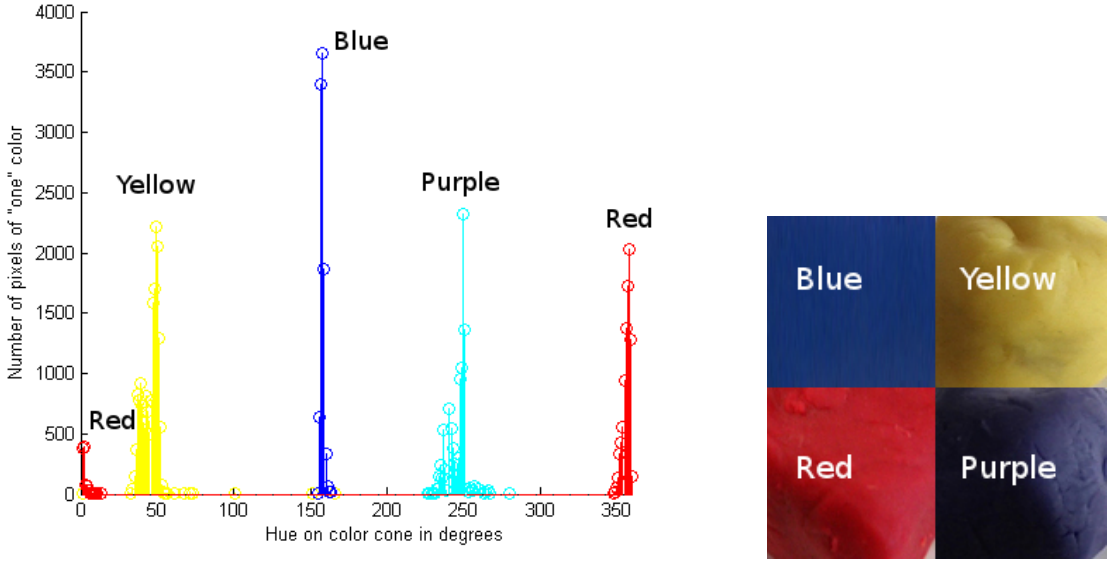


Figure 5.31: A HSV cone from [31] that shows all channels. The circle illustrates, why there is no maximum or minimum *hue* value. All colors are 2π periodical, whereas red is at a *hue* value of around 0 or 2π.

The vision is also able to detect red, like any other color. If one of the borders is exceeded, the detection includes the overlap on the other border side. In this case this corresponds to two color detections for one color, where two masks are combined with a bitwise OR. Instead of purple, the color red was chosen for the tokens, because the game board received a blue stripe to support localization. The color red is more different from blue than purple. Although we talk about colors, the robot does not know any color. It just looks for color values that are in a range of the values, which are initialized at the calibration step. The distances on the HSV cone between the different token colors have been visualized in a histogram in Figure 5.32a, whereas the source image for the histogram is shown in Figure 5.32b. The source image contains the blue stripe and a token of yellow, red and purple color.



(a) Histogram

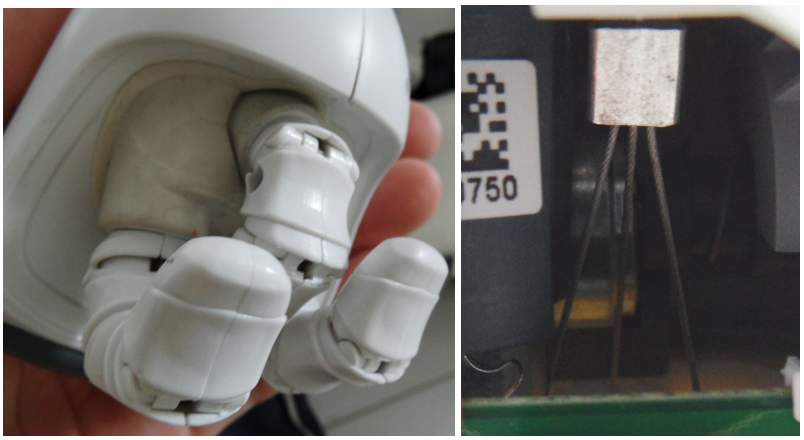
(b) Source image

Figure 5.32: The histogram of Figure 5.32a shows the game token colors and the blue stripe color in the HSV color space in a resolution of 360 bins. The left side at bin zero represents a hue in the color cone of zero, while the right side of bin 360 represents 2π. The overlap of the red color is also visible. Figure 5.32b shows the source image of the histogram.

Different kind of tokens for the robot to grasp were tested. At first ordinary tokens from a human Nine Men's Morris game were used. In general these tokens have a very low height and are difficult to grasp with Nao autonomously. The next token was ball-shaped, with the intention of having a uniform surface to detect and grasp. Although they have no sharp borders, which is good for detection, the ball-shaped tokens tend to roll away or slip out of the hand, if only grasped with two fingers. So in the next test, a cuboid wood token was used. The cuboid-shape does not roll away and was also graspable with two fingers, but it has edges that result in shadows and it can tilt, if not placed properly.

All fingers of one hand (see Figure 5.33a) are simultaneously controlled by one wire rope in each hand, as shown in Figure 5.33b, which is a little bit fragile. A material was required to preserve the hand functionality. Therefore tokens out of foam were manufactured. The tokens performed well on a grasp action, but if the token is placed, it could jump away like a rubber ball.

The only experienced disadvantage of modeling clay is that it air-drys quickly. After each use the tokens need to be put back in a plastic box or bag. A cold temperature supports the lifetime of modeling clay tokens.
The final 2×9 game tokens already have been shown in Figure 4.1. They are not all of equal size but nevertheless the robot is able to play with them, because the recognition has enough tolerances to deal with different sizes and distortions.



(a) Nao's fingers that are controlled by one wire rope.

(b) Wire rope mechanic to open and close all fingers.

Figure 5.33: The pincer fingers and the corresponding wirerope to grasp something

5.8.1 Production of the Game Tokens

The amount of nine tokens requires approximately the following ingredients.

- $\frac{1}{4}$ cup of water
- $\frac{1}{8}$ cup of salt
- $\frac{1}{2}$ table spoon of alum[1] [42]
- some color [2]
- 1 table spoon of oil
- and a $\frac{3}{4}$ cup of flour

For the best result the first four ingredients need to be mixed in a pot and boiled. Then the last two ingredients need to be added. Afterwards the mixture has to be removed from the pot and pugged until a homogeneous mixture is received that feels like plastic modeling mass. The flour generates the basic compound of the modeling clay, while the alum (shown in Figure 5.34), oil and salt influences the consistency. Especially the alum and oil together with an air-proof storage keeps the modeling clay soft. The cold mass can be stored in a plastic bag in the fridge with an approximate durability of one year depending on the usage.

[1] Alum: is a special chemical compound that consists of hydrated potassium aluminum sulfate and can be bought in a pharmacy

[2] Colored powder of easter egg color has been used, mixed together with the fluid to receive colored tokens in the end. Note that in contrast to the easter egg color that colored the mixture very well, colored juices will not work. They will just make the mixture stickier and do not color the tokens very well.

Figure 5.34: Alum powder in its solid form, which was used to create modeling clay.

5.9 Other Packages

In order to solve the complete task, our framework uses other ROS-packages that provide tools and functions. Following packages have been used:

- **roscpp** "is a C++ implementation of ROS. It provides a client library that enables C++ programmers to quickly interface with ROS Topics, Services, and Parameters." [36].
- **rospy** "is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters. The design of rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS. It is also ideal for non-critical-path code, such as configuration and initialization code." [36]
- **opencv2, cv_bridge and image_transport** are interfaces to the OpenCV library and easy image handling [36, 32].
- **sensor_msgs, geometry_msgs and visualization_msgs** are predefined ROS messages that include points clouds, vectors, markers etc. [36].
- **tf** is the transformation package of ROS, which is required for broadcasting and listening to it [36].
- **camera_calibration_parsers** are used to parse configuration text files that contain camera parameters [36].
- **AR ToolKit** and **resource_retriever** are required for the AR ToolKit to recognize the marker. Although the AR ToolKit is no ROS package, we created one that contains the its framework [5].
- **arm_navigation_msgs**, **nav_msgs** and **kdl** are required for the kinematics.
- **planning_environment** "is a library/ROS node that allows users to instantiate robot models and collision models based on data from the parameter server with minimal user input." [36]
- **robot_state_publisher** publishes the joint states of the robot to the tf [36].
- **rviz** is a visualization environment that supports cameras, transformations, points clouds and predefined visualization messages to model the robot and its environment [36]. It is used to visualize and debug the whole framework. Once rviz has been configured for the project, this configuration can be saved to a file. Additionally the virtual environment is drawn in the camera, which is very helpful for debugging.

Chapter 6

Experimental Evaluation

This chapter presents an evaluation of our experimental results. Explanations to specific results are discussed in Chapter 7. The overall evaluation is similar to the work of [28], because this master project was developed in cooperation and the components can be tested in combination of the motion and vision modules only. For example, to get a game state, the robot needs to move the head to watch specific positions on the board. However the discussion is focused on the vision tasks.

The decision of the color of the game tokens is discussed in Section 5.8. In all tests the color calibration detected the correct colors. However it also depends on the precision of the user that places the tokens, which are used for calibration. If the tokens are not located at the ideal positions at the initialization, the calibration cannot work.

6.1 Network Traffic

The network traffic test evaluates the bandwidth usage of all nodes that communicate with the robot over the network. Regarding our framework, the average network traffic was measured with a system monitor in Ubuntu. The results can be seen in Table 6.1.

Node	Mean transfer rate	Percent of usage
nao_get_data.py	5 KB/s	6.5 %
nao_cam.py	50 KB/s	64.9 %
nao_motion_control.py	2 KB/s	2.6 %
nao_tf.py + world_model	20 KB/s	26.0 %

Table 6.1: Average network traffic between the NaoQi and ROS

6.2 Token Placement

In the placement sequence the robot raises its arm, opens its fingers and waits for a human to hand over a token of its color. The token has to be recognized, the fingers have to be closed and the hand has to be moved to the target ideal token position without touching anything. When the hand is at the target position, the fingers open again, release the token and the arm retracts. Due to the shape of some tokens, they could roll, tilt or just fall out of the hand. Thus the robot checks, if the placed token is on the desired position.

The recognition has not been mentioned in the table, because all tokens have been recognized correctly before the placement procedure. The arm position before the placement can be seen in Figure 6.1. The fingers are already closed a little bit, because the robot has recognized the token.

Figure 6.1: Pre-placement position of the hand, after the token has been recognized and the fingers have been closed, but before the hand moves to the target position.

Figure 6.2 shows the hand at a target token position on the middle square at the moment when the token was placed. One can also see that the token will tilt a little bit. The reliability of the placement procedure has been evaluated with 30 placements of tokens on the board. We evaluated the accuracy of the token placement. Therefore we distinguished between a success or fail of the placement process. For a successful placement, a token has to be placed inside its target mask area, next to the ideal position. If a token is not placed next to its ideal position, but nevertheless in its mask, it could be also classified as a fail placement. On the inner square there is only a little tolerance to place a token. Not accurately placed tokens could be misclassified in subsequent game turns.
Additionally we rated how many of the placements were repaired, after the token could not be confirmed on its target position. The results can be found in Table 6.2.

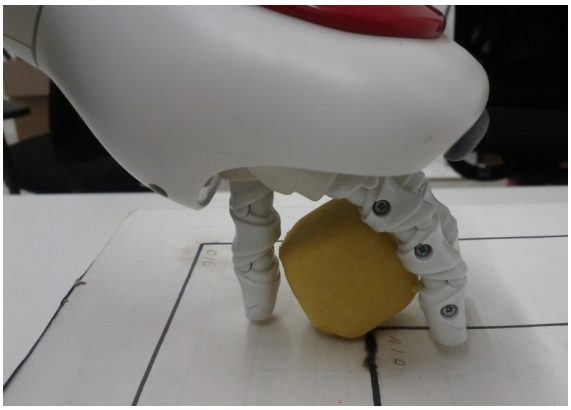


Figure 6.2: Placement on the middle square

		Repair place		Σ	%
		Success	Fail		
Place	Success	28	0	28	93.33
	Fail	1	1	2	6.66
Σ		29	1	30	
%		96.66	3.33		

Table 6.2: Evaluation of token placements

6.3 Token Grasp

The grasp procedure includes the localization of the robot and the real token position. The robot has to calculate its arm angles to move the opened hand to the target real token position to grasp the token. When the hand is at the target position, the fingers are being closed to grasp the token and the hand is raised to a predefined position for visual confirmation.
A grasp on the inner square is shown in Figure 6.3. One can see that the fingers are very close to the marker. Another grasp from different views to the middle position of the middle square can be seen in Figure 6.4. The predefined hand position for the visual confirmation can be seen in Figure 6.5a. The grasp in the last mentioned figure was a perfect grasp, as the token between the three fingers has no possibility to fall out of the hand. Another example, where the token has been grasped using two fingers only, is shown in Figure 6.5b. Although this is a valid grasp, it becomes a challenge, when the token is lost during motion actions.
The grasping and recognition were evaluated in 30 grasps. A token that was not grasped or lost anywhere, was classified as fail. The recognition fails, if a token is between two finger and could not be detected or if a token is detected, when the hand is empty. This test was important to verify the accuracy of the real token position, calculated after the rectification step of the blob classification. The results are shown in Table 6.3.

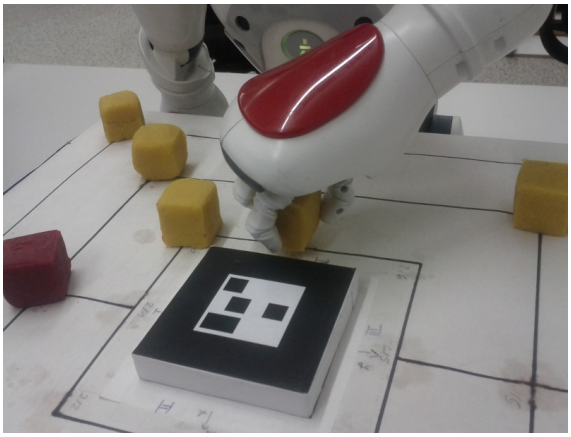
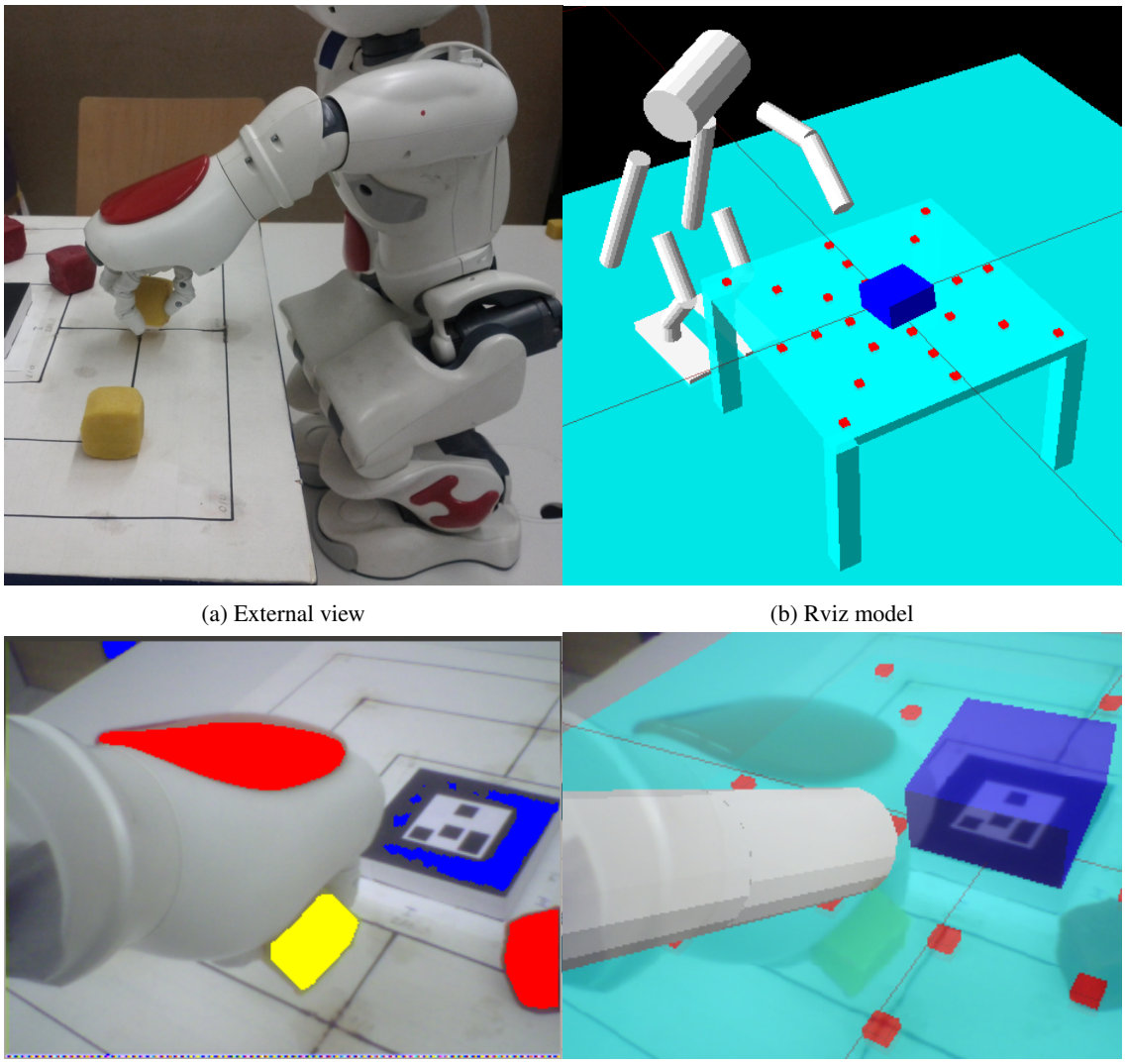


Figure 6.3: A grasp of a token on the inner square

		Recognition		Σ	%
		Success	Fail		
Grab	Success	18	2	20	66.66
	Fail	10	0	10	33.33
Σ		28	2	30	
%		93.33	6.66		

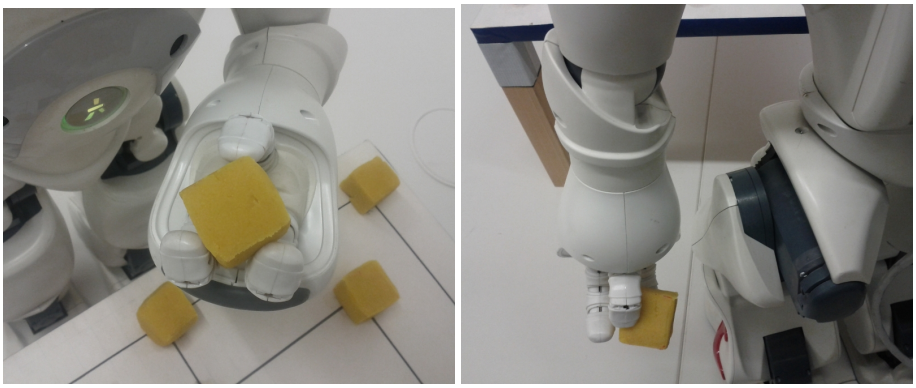
Table 6.3: Evaluation of token grasps



(a) External view (b) Rviz model

(c) Internal camera view (d) Internal camera view with rviz overlay

Figure 6.4: All four figures show the grasping procedure of a middle position on the middle square from different views.



(a) Perfect pose of a successful grasp (b) Successful, but not safe grasp

Figure 6.5: Different token poses inside the fingers

6.4 Token Classification

The next test evaluates the reliability of the rectification and token classification. The robot has to segment the tokens from the background, classify it to a strategic position and combine it with multiple observations to one game state. Sometimes tokens do not lie on their expected position, which may lead to a fail detection of single positions. The game state procedure has to classify different game board occupations with random amounts of game tokens. Figure 6.6 shows a natural game state in the movement phase from an external camera. An artificial setup from an external view can be seen in Figure 6.7. The detected game state of the last mentioned figure can be found in Figure 6.8. In contrast to the figures taken by an external camera, Figure 6.9 represents a game state in the rviz model. We evaluated how many tokens and complete game states are classified correctly. Empty positions do not contribute to the correct classification, only tokens that are classified correctly. However, if an empty position is classified as a token, it contributes to the misclassification. The evaluation can be found in Table 6.4.

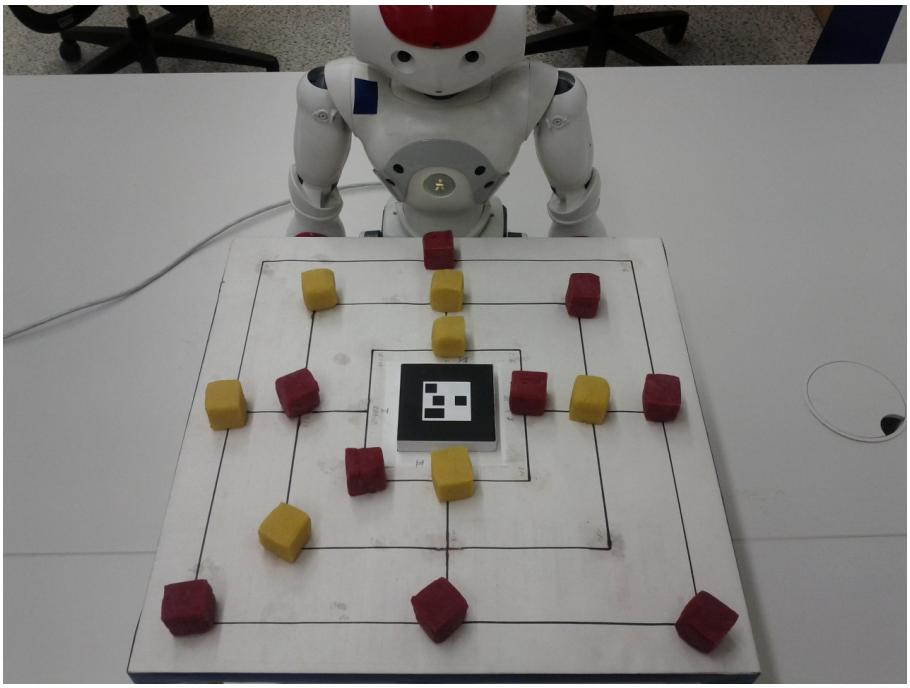


Figure 6.6: External view to a possible game state during a game

Number of tests		38	%
Number of classified tokens	Correct	359	98.63
	Wrong	5	1.37
Game states without any misclassification	Correct	33	86.84
	Wrong	5	13.16

Table 6.4: Evaluation of token and game state classification

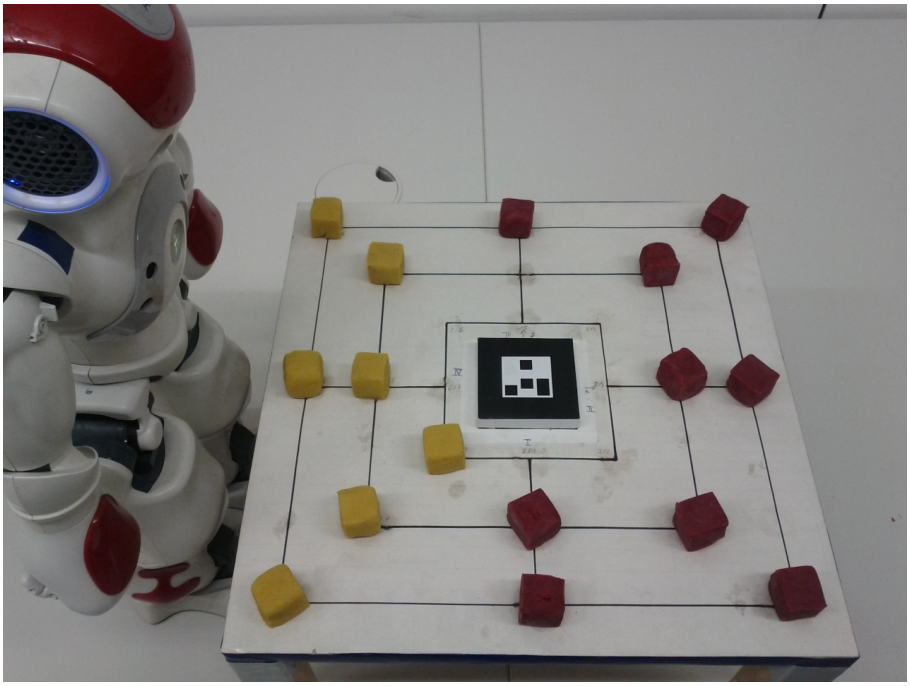


Figure 6.7: External view to an artificial game state setup

```
Game State:
1--------2--------2
|        |        |
|  1-----0-----2  |
|  |     |     |  |
|  |  0--0--0  |  |
|  |  |     |  |  |
1--1--0  M  0--2--2
|  |  |     |  |  |
|  |  1--0--0  |  |
|  |     |     |  |
|  1-----2-----2  |
|        |        |
1--------2--------2
```

Figure 6.8: Detection result printed to the console for the game state, which is shown in Figure 6.7

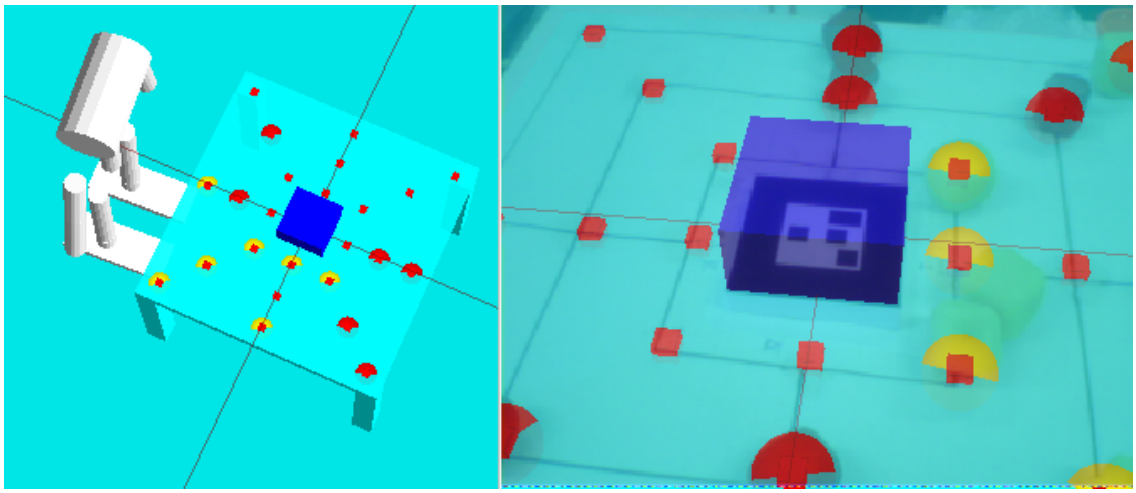
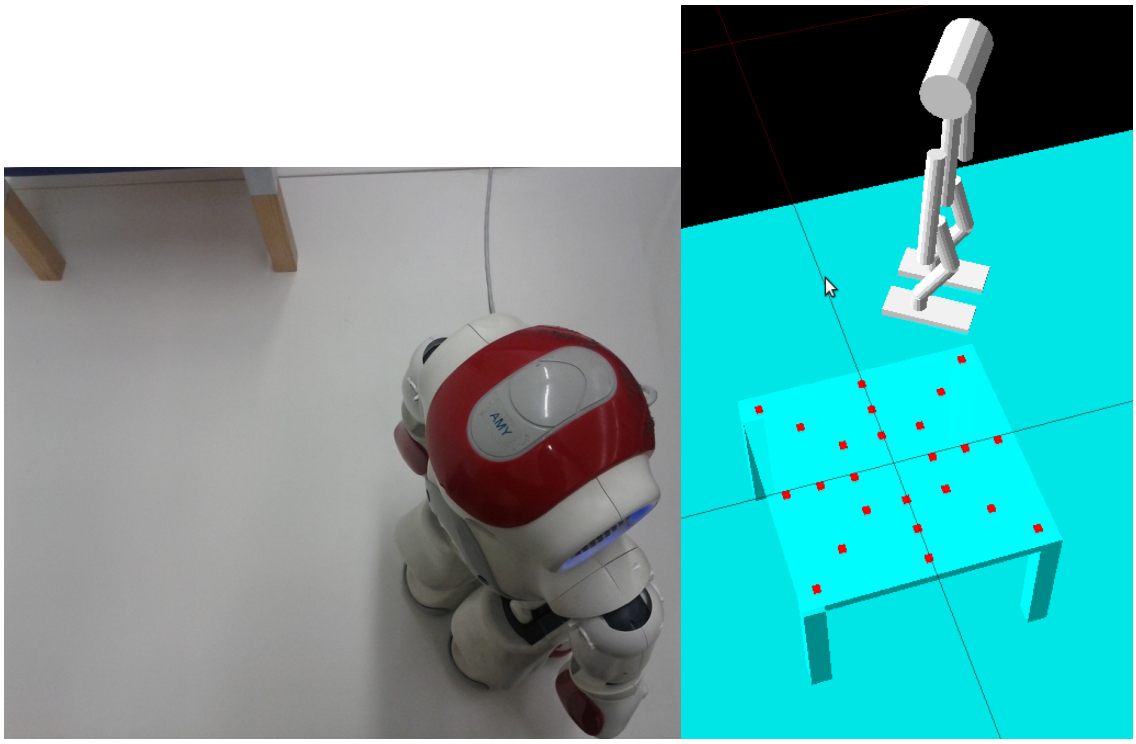


Figure 6.9: Game state representation in rviz with the robot model, table, marker and game tokens on the left. The yellow and red balls represent the recognized tokens and therefore the game state.

6.5 Walk and Localization

Board positions on other table sides, require the robot to move the torso around the game board to manipulate a token. A walk includes the save movement from, around and towards the table using the internal odometry for localization, the visual odometry to correct rotations and a path planner for the path around the board. The relocalization after the walk around and before the walk towards the table is one of the most difficult parts, because the marker is far away and sometimes occluded by tokens.
In this evaluation, 23 walks around a quarter of the table have been evaluated. All walks, where the robot moved correctly without touching anything, have been classified correctly. Additionally we added a category for slight collisions, because sometimes the robot streaked the table without any change for the game situation. Walks fail, if the robot bumps into the table, fails to relocalize, moves too far or short. Most of the time there is a reason for a fail walk, like a obstacle on the ground, a tensed network cable or hot joints. In these case we did not rate the walk. Figure 6.10 shows the robot in a corner position, which was passed during a walk. When the robot is located in such a position, it cannot see the marker. On rotations around its vertical axis, deviations of up to ten degrees are possible, depending on the angle to rotate. On other undergrounds the maximum deviation may differ from our tests. The visual odometry works as expected (see Figure 6.11), as long as the background that is captured by the top camera does not change between the rotation of the head and torso. If the background changes completely, as shown in Figure 6.12, not enough matches can be found to make an assumption about the rotation. In this case, the odometry informs the walk planner that no correction can be applied and the robot has to rely on its internal odometry. The walk evaluation can be found in Table 6.5.



(a) External view (b) Rviz representation

Figure 6.10: A walk around a corner of the board. The robot is far away from the board, which is visible in the top left corner of Figure 6.10a. The robot cannot see the marker in this pose.

		Walk			Σ	%
		Success	Slight collisions	Fail		
Marker	Found	10	4	0	14	60.86
	Lost	9	0	0	9	39.13
Σ		19	4	0	23	
%		82.61	17.39	0		

Table 6.5: Evaluation of walks around the board

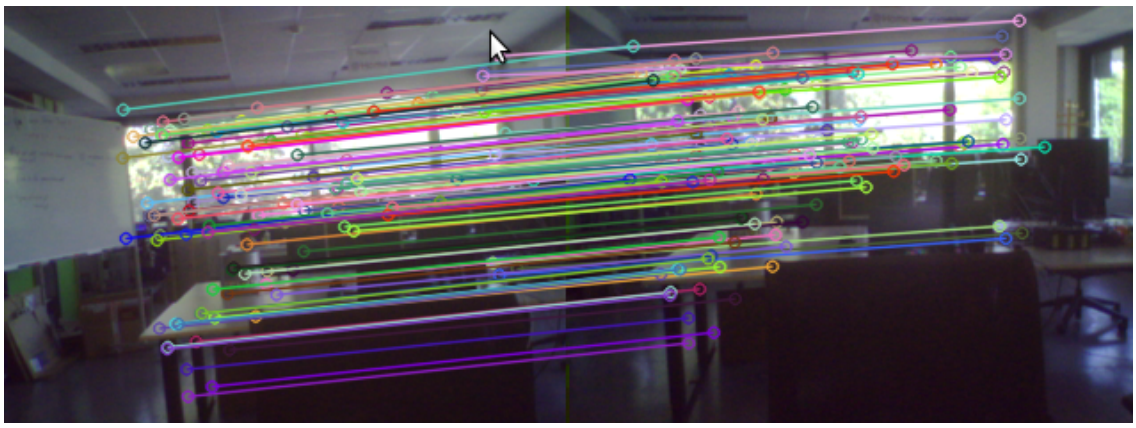


Figure 6.11: Visual odometry: Both images after rotation with enough visualized matches to compute the rotation angle deviation.

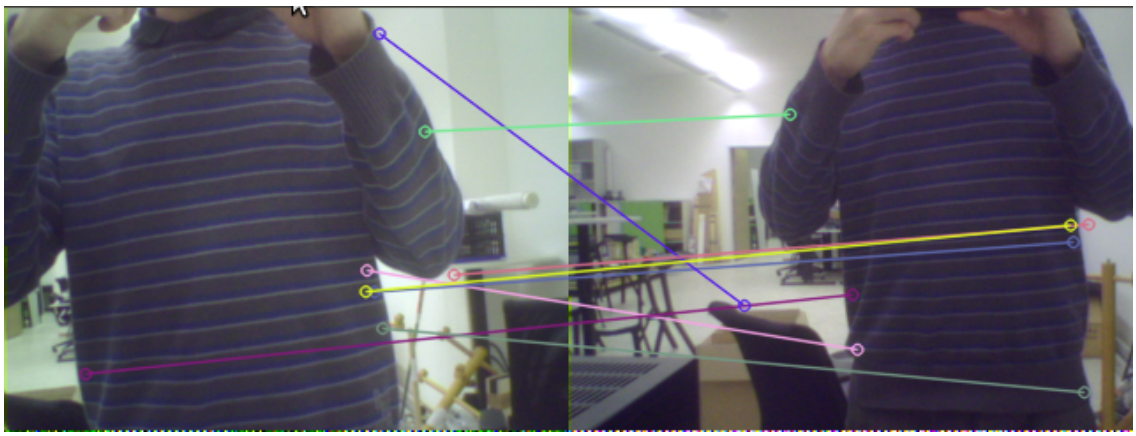


Figure 6.12: Visual odometry: Both images after rotation with too little matches. No reliable assumption of the rotation deviation is possible.

Chapter 7

Conclusion

This thesis presents a general architecture for playing a board games autonomously with a humanoid robot. We faced and solved many challenges that arose during game play, like robust localization and classification of game tokens and accurate position estimation with a single camera. We showed a working implementation, which does not require any external sensors to successfully play Nine Men's Morris.
In the end, when we tried to play whole games, we had a severe robot specific problem. During the move of the opponent player the robot has some time to relax its joints. This time is limited by the time the user needs to perform a move, but is essential to prevent the joints from overheating. On continuous usage, with only little time to relax, Nao's motors are overheating and they lose stiffness. This leads to instable walk procedures, where the robot just breaks down or falls on the ground. We thought about changing the strategy to lower the board where Nao's has to bend over, but where it is able to acquire the game state while sitting to relax its joints as in the approach of [25]. Additionally a lower game board improves the angle from the camera to the marker. It would be one way to cope with stability issues, but the board height also influences the workspace of possible positions to manipulate tokens. Moreover the robot could not sit with its legs below the table and therefore the distance to the game board increases. Possibly there will arise other challenges, we have not taken into consideration yet.
The game engine framework has been included in our framework and was tested with a dummy database. At the moment a simulator is used for playing, because the computation of the full database is still running. The following sections discuss the experimental results of the individual tasks.

7.1 Network Traffic

The reason why this was measured is that we do not know why the wireless connection was very weak. We assumed that one node consumes the network bandwidth, but the traffic evaluation has disapproved that. When the robot was connected wireless, movement commands were executed a few seconds delayed and recorded images arrived approximately 10 seconds later. Even with deactivated vision, the delay was remarkably worse than a wired connection. Recently we used another wireless router next to the robot, which improved the connection and solved the delay issue. However, the transmission over the Ethernet cable is three times faster. As table 6.1 shows, most network traffic is produced by transmitting the camera images from the robot to our computer. The other major part of the traffic is created by a script that periodically requests all joint values, which are required for the localization and kinematic.

Although a wireless connection is convenient, we faced an issue with delayed camera images. The vision tasks are synchronized by time stamps, which are saved in each image. If a task requires an updated image, it waits until it gets an updated one. The timestamp of each image is set in the camera node, when the transmitted image arrives at our computer. It would better to set the time stamps when the images are

taken. The NaoQi also sets a timestamp at the image on capture, but as the robot has an external system and clock, it is not synchronized with our framework yet.

7.2 Place

As table 6.2 shows, the placement procedure works quite well, even including game board side changes, the target position always was found correctly. Furthermore the requested token always was recognized immediately after it has been placed in the hand. Sometimes it happens tokens roll away or tilt.
In the 30 test cases the tokens rolled out of the target mask once, into another mask, where a token of the same color was already occupying this position. The robot noticed that the token was not on the target position, scanned the whole game board and detected the missing token in the neighboring mask. It automatically grasped this token successfully and placed it again on its destination.

The one fail case was a placement in the correct target mask, but the token tilted on release, so it was "far" away from its ideal position (see Figure 7.1). This concerned a token on the inner square, which means that there is only very little tolerance to the next token on that square. Although the game state was classified correctly from the same side, this token was not found from another table side, when it was occluded by another token. Hence this placement was classified as fail.

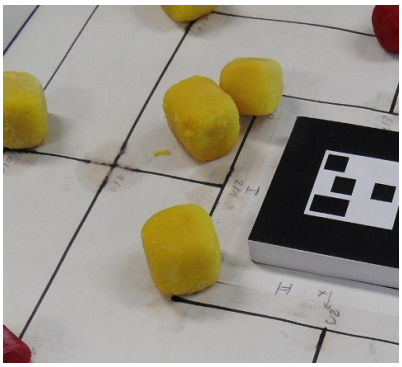


Figure 7.1: A misplaced token on the inner square

7.3 Grab

Table 6.3 shows that there are some issues concerning the grasp of a token. In the test setup, 33,3% of them were not grasped correctly, because the three-dimensional position was not accurate enough or the thumb missed the token. Even a slight uncertainty of a few millimeters results in a fail grasp, because the fingers will slip on the border of the token and push it away. In two cases the token was grasped correctly, but was not recognized in the hand. This could happen if the token dropped out of the hand, because the grip was not firm enough or the token was not correctly arranged inside the hand. The first and easiest issue has been resolved in the meantime by grasping a little bit firmer. Sometimes the tokens are grasped with only two fingers as shown in Figure 7.2. If the robot opens its fingers, the token will fall down in front of the robot. In contrast to a grasp with the thumb and the outer finger, where the token will roll into the palm. It will stay there until a human player removes it. This could be solved by evaluating the area, which is occupied by the token in the hand. In case the token is close to the robot, which refers to the inner finger, the robot will rotate the wrist a little bit, so that the token will also roll into the palm instead of falling down.

Figure 7.2: Unsafe grasp with two fingers

7.4 Get Game State

The procedure to acquire the game state works reliable, however there are some special cases where all tokens could be detected, stated in Table 6.4. In two cases, the unrecognized token was a misplaced in the inner square. From the side where the token was placed, it could be recognized but when the robot walked to around one corner, this token was too far away from its expected position and the position was marked as unoccupied. All other positions, except some distant tokens, were classified correctly. At the upmost, only one token was classified falsely in each game state. The game state recognition can be further improved by a voting strategy for all positions and players, also regarding different view angles. The speed of the recognition could be increased by planning a more intelligent trajectory. Both ideas require an intelligent handling for incomplete seen masks of a position, which has not been solved yet.

7.5 Walk

Lately we discovered that the walk procedure works much better without the connected power cable. We classified some walk procedure as slight fail in Table 6.5, because the robot touched the game board slightly, without any harm or game change. The “advantage“ of a slight fail walk is that the robot is closer to the board, so the marker can be found quickly to relocalize. In other cases the robot walked around the board perfectly, but did not find the marker, because from its perspective the marker was occluded by tokens. As the robot did not find the marker, it looked about for a blue stripe in front of it and walked towards it, leaving enough space for not touching the game board. When the robot was closer to the board, the marker was found and it relocalized. When the robot was relocalized, it stepped to the table correctly. The visual odometry also worked very well to correct the rotation of the torso. However in Figure 6.12, I move in front of the robot, which results in different images and no reliable assumption can be made, because there are too little matches. If the inner circle is completely occupied, it gets very difficult to detect the marker, because the black border of it also requires a white adjacent region for detection. This is sometimes not possible if the player of the darker color has tokens in the inner square.

7.6 Future Work

There are several possibilities to improve our framework, as we want to have a less restrictive setup. Even though this project used a marker for the localization, further developments are necessary to show up that it is possible to play the game without a marker using an adapted SLAM as proposed in [14] or [27]. The drawback of such a mapping and self-localization approach is that the camera has to be active all the time, tracking the movement. The tracking procedure consumes processing power and has difficulties to handling motion blur. There are strategies to find a trade off between movement speed and motion blur, proposed in [24].
We would also like to reduce the human interaction that is required for game play. Instead of handing over tokens that are going to be placed or removed, the robot should take the tokens out of a box next to the game board. If we improved the range of the ability to grasp tokens lying in the middle of the game board, the robot would only need to move at maximum around one corner of the table for one manipulation. This would also be one possibility speeding up the framework. Additionally for some token moves, it would be possible to swap the token between the hands to save a walk around the board.
We thought about an automatic color calibration at startup. Instead of the two predefined positions, which are used to initialize the player colors, the robot could take one color that is distinctive from the game board surface color after the turn of the human player. Its own color can be calibrated, when the robot waits for a token being handed over by the human player. When the color camera image that watches the fingers change, the robot knows that something has to be in its hand, which would be a token of the robot's player color.
As a next step we thought about synchronizing the time of the robot with our external system, by measuring the latency and time difference. This way we could set the correct time stamp for the images, when they were taken and the framework would be delay independent. Then the wireless connection could be used too. Future approaches should also use an online obstacle detection to detect dynamically changing environments, as well as a native implementation on the robot. Finally the architecture should by expanded to handle a variety of board games. Board games like chess, where the game tokens are inhomogeneous, so we believe will be the next challenge.

Bibliography

[1] AICHHOLZER, O., DETASSIS, D., HACKL, T., STEINBAUER, G., AND THONHAUSER, J. 2010. Playing pylos with an autonomous robot. In *IROS*. IEEE, 2507–2508. (Cited on page 19.)

[2] ALDEBARAN. Homepage - Aldebaran Robotics. http://http://users.aldebaran–robotics.com. [Online; accessed 21-September-2012]. (Cited on pages 10 and 50.)

[3] ALDEBARAN. Nao Documentation - Video Device. https://developer.aldebaran–robotics.com/doc/1–12/nao/hardware/video.html. [Online; accessed 13-October-2012]. (Cited on page 12.)

[4] ALDEBARAN. 2012. Nao-User-Guide-1.6.13. http://developer.aldebaran–robotics.com/doc/1–12/. [Online; accessed 21-September-2012]. (Cited on pages 5, 10, and 50.)

[5] ARTOOLKIT. 2012. http://www.hitl.washington.edu/artoolkit. [Online; accessed 21-September-2012]. (Cited on pages 14, 53, and 73.)

[6] BARTZ, M. 2001. Quaternionen. University Koblenz-Landau, 3–11. (Cited on page 46.)

[7] BAY, H., ESS, A., TUYTELAARS, T., AND VAN GOOL, L. 2008. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst. 110,* 3 (June), 346–359. (Cited on page 6.)

[8] BOCK, S. AND KLÖBL, R. Nine Men's Morris Framework. http://www.ist.tugraz.at/robotics/bin/view/Main/PlayingNineMenMorrisFramework. [Online; accessed 13-October-2012]. (Cited on page 2.)

[9] BOUGUET, J.-Y. 2008. Camera Calibration Toolbox for Matlab. http://www.vision.caltech.edu/bouguetj/calib_doc/. [Online; accessed 21-September-2012]. (Cited on page 61.)

[10] BRADSKI, G. AND KAEHLER, A. 2008. Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly Media, 407–412. (Cited on page 67.)

[11] BRUCE, J., BALCH, T., AND VELOSO, M. 2000. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '00)*. Vol. 3. 2061 – 2066. (Cited on page 18.)

[12] CHIUSO, A., FAVARO, P., JIN, H., AND SOATTO, S. 2000. 3-D Motion and Structure from 2-D Motion Causally Integrated over Time: Implementation. In *In IEEE Trans. Robotics and Automation*. 734–750. (Cited on page 16.)

[13] DAHM, I., DEUTSCH, S., HEBBEL, M., AND OSTERHUES, A. 2004. Robust color classification for robot soccer. In *in RoboCup 2003: Robot Soccer World Cup VII, ser. LNCS 3020*. Springer, 677–686. (Cited on pages 17 and 47.)

[14] DAVISON, A. J. 2003. Real-Time Simultaneous Localisation and Mapping with a Single Camera. In *Proceedings. Ninth IEEE International Conference on*. IEEE Computing Society, Oxford, UK, 1403 – 1410 vol.2. (Cited on pages 16, 17, and 86.)

[15] DAVISON, A. J. AND MURRAY, D. W. 2002. Simultaneous Localization and Map-Building Using Active Vision. *IEEE Trans. Pattern Anal. Mach. Intell. 24,* 7 (July), 865–880. (Cited on page 17.)

[16] DRYANOVSKI, I., MORRIS, W., AND DUMONTEIL, G. ROS wrapper for the ARToolKit. http://www.ros.org/wiki/ar_pose. [Online; accessed 13-October-2012]. (Cited on page 14.)

[17] DURRANTWHYTE, H. F., DISSANAYAKE, G., AND GIBBENS, P. W. 2000. Toward deployment of largescale simultaneous localisation and map building (SLAM) systems. In *In Robotics Research, The Ninth Int. Symposium*. Springer-Verlag, 161168. (Cited on page 16.)

[18] FITZGIBBON, A. W. AND ZISSERMAN, A. 1998. Automatic Camera Recovery for Closed or Open Image Sequences. In *European Conference on Computer Vision*. Springer-Verlag, 311–326. (Cited on page 16.)

[19] GAARDER, J. 2012. Sophie's World. http://www.wattpad.com/80326–sophie%27s–world–jostien–gaarder?p=137#!p=137. [Online; accessed 26-September-2012]. (Cited on page 2.)

[20] HARAHAP, D. A., PRABUWONO, A. S., AND ABDULLAH, A. 2011. Illumination normalization methods for object recognition in robot soccer vision. In *Pattern Analysis and Intelligent Robotics (ICPAIR), 2011 International Conference on*. Vol. 1. 109 –113. (Cited on page 17.)

[21] HARTLEY, R. AND ZISSERMAN, A. 2003. Multiple View Geometry in Computer Vision. Cambridge University Press, New York, NY, USA, 33–35. (Cited on pages 64 and 65.)

[22] HÖLL, T. AND PINZ, A. 2011. Vision-based grasping of objects from a table using the humanoid robot Nao. Austrian Robotics Workshop. (Cited on pages 18 and 19.)

[23] HORNUNG, A. Nao - ROS Driver. http://www.ros.org/wiki/Robots/Nao. [Online; accessed 13-October-2012]. (Cited on pages 24 and 54.)

[24] HORNUNG, A., BENNEWITZ, M., STRASDAT, H., HORNUNG, A., BENNEWITZ, M., AND STRASDAT, H. 2010. Efficient vision-based navigation Learning about the influence of motion blur. *Journal of Autonomous Robots*, 137–149. (Cited on pages 16 and 86.)

[25] HUMAROBOTICS. 2012. NAO plays... Connect 4. http://www.generationrobots.com/site/program–nao–robot/index_en.html. [Online; accessed 21-September-2012]. (Cited on pages 20 and 83.)

[26] KATO, H. AND BILLINGHURST, M. 1999. Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System. In *Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99)*. San Francisco, USA. (Cited on page 53.)

[27] KLEIN, G. AND MURRAY, D. 2007. Parallel Tracking and Mapping for Small AR Workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*. Nara, Japan. (Cited on pages 17 and 86.)

[28] KLÖBL, R. 2012. Autonomous Grasping and Path-Planning for a Humanoid Robot Playing Nine-Men's-Morris. (Cited on pages i, ii, 24, 27, 29, 35, 36, 46, and 74.)

[29] LEONARD, J. J. AND FEDER, H. J. S. 2000. A computationally efficient method for large-scale concurrent mapping and localization. In *In Robotics Research*. Springer-Verlag. (Cited on page 16.)

[30] LIEMHETCHARAT, S., COLTIN, B., AND VELOSO, M. 2010. Vision-Based Cognition of a Humanoid Robot in Standard Platform Robot Soccer. In *Proceedings of the 5th Workshop on Humanoid Soccer Robots @ Humanoids 2010*. IEEE Computing Society, Pittsburgh, PA, USA, 47–52. (Cited on page 16.)

[31] MICROSOFT. HSV Cone. http://msdn.microsoft.com/en–us/library/windows/desktop/aa511283.aspx. [Online; accessed 21-September-2012]. (Cited on page 70.)

[32] OPENCV. 2012. OpenCV Library. http://opencv.org. [Online; accessed 13-October-2012]. (Cited on pages 56, 65, 66, 68, and 73.)

[33] PARK, E., KONG, H., LIM, H.-T., LEE, J., YOU, S., AND DEL POBIL, A. P. 2011. The Effect of Robot's Behavior vs. Appearance on Communication with Humans. In *Proceedings of the 6th international conference on Human-robot interaction*. HRI 2011. ACM, New York, NY, USA, 219–220. (Cited on pages 20, 21, and 22.)

[34] REGENFELDER, O. 2010. Implementation of Games in a Generalized Framework for Solving Combinatorial Games using the State Space. In *Faculty of Computer Science, Graz University of Technology*. (Cited on pages 25 and 47.)

[35] ROBOCUP. http://www.robocup.org. [Online; accessed 21-September-2012]. (Cited on page 22.)

[36] ROS-WIKI. 2012. Robot Operating System Wiki. www.ros.org/wiki/. [Online; accessed 17-September-2012]. (Cited on pages 6 and 73.)

[37] SHI, J. AND TOMASI, C. 1994. Good Features to Track. In *IEEE Conference on Computer Vision and Pattern Recognition*. 593–600. (Cited on page 17.)

[38] SUN, Y., XIONG, R., ZHU, Q., WU, J., AND CHU, J. 2011. Balance motion generation for a humanoid robot playing table tennis. In *Humanoids*. IEEE, 19–25. (Cited on page 19.)

[39] TERRILLON, J.-C., SHIRAZI, M., FUKAMACHI, H., AND AKAMATSU, S. 2000. Comparative performance of different skin chrominance models and chrominance spaces for the automatic detection of human faces in color images. In *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on*. IEEE Computing Society, Kyoto,Japan, 54 – 61. (Cited on page 17.)

[40] THRUN, S., FOX, D., BURGARD, W., AND DELLAERT, F. 2000. Robust Monte Carlo Localization for Mobile Robots. *Artificial Intelligence 128,* 1-2, 99–141. (Cited on page 16.)

[41] TOMLINSON, S. 2012. Human-Robot Interaction: Dominoes Player. http://mipal.net.au/video.php. [Online; accessed 21-September-2012]. (Cited on page 20.)

[42] WIKIPEDIA. Alum. http://en.wikipedia.org/wiki/Alum. [Online; accessed 21-September-2012]. (Cited on page 72.)

[43] WIKIPEDIA. Quaternions. http://en.wikipedia.org/wiki/Quaternion. [Online; accessed 13-October-2012]. (Cited on page 6.)

[44] WIKIPEDIA. YUV. http://en.wikipedia.org/wiki/YUV. [Online; accessed 13-October-2012]. (Cited on page 5.)

[45] ZHANG, X., FRONZ, S., AND NAVAB, N. 2002. Visual marker detection and decoding in AR systems: a comparative study. In *Mixed and Augmented Reality, 2002. ISMAR 2002. Proceedings. International Symposium on*. Vol. 1. 97–106. (Cited on page 53.)

[46] ZHANG, Z. 1999. Flexible camera calibration by viewing a plane from unknown orientations. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. Vol. 1. 666–673. (Cited on page 61.)