Freie Universität Berlin
Department of Mathematics and Computer Science

Diploma Thesis

Development of a Multi-Level Sensor Emulator for Humanoid Robots

Steffen Heinrich

23rd January 2012

Supervisor: Prof. Dr. Raúl Rojas
Advisor: Dr. Hamid Reza Moballegh

Declaration of Academic Honesty

I hereby declare that this diploma thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Berlin, 23rd January 2012

Steffen Heinrich

Abstract

For the purposes of this diploma thesis, simulation software for autonomous soccer playing humanoid robots was developed and integrated into the software eco-system of the FUmanoids. In recent years humanoid robot control software has become more complex and, as a result, essential components such as computer vision, motion or behavior control have required improved quality assurance methods.

The software developed provides access to emulated sensor devices, such as cameras, actuators, inertial measurement units and pressure sensors. Furthermore, virtual sensors are able to generate information with additional meta data such as extracted object positions instead of a raw camera image, allowing multiple levels of data abstraction to be received by the control software. This means that the current robotic hardware model is no longer explicitly needed. In addition, the software provides access to a wide range of dynamic libraries simulating physical feedback.

Multi-level testing methods can be applied to achieve higher quality in each module and module tests are no longer strictly linked to the use of robotic hardware. As a result of this identical test cases are now reproducible and can be applied to the same module several times.

Retrospectively, the software had a positive effect on the development of the behavior control module for RoboCup 2011. It was also well integrated in the test sessions of the recently developed localization module and experimental evidence has shown that the simulation is capable of emulating a virtual counterpart to a Dynamixel servo motor.

Contents

1 Introduction

This thesis presents recently-developed simulation software for the humanoid soccer playing robots "FUmanoids". This simulation package integrates easily with the existing software eco-system and removes the need for physical robotic hardware during testing sessions. The software is capable of emulating sensor devices as well as simulating the physical forces that act on all of the body parts and joints of a virtual robot.

In recent years, humanoid robot control software has become more complex due to the increase in the performance of the embedded systems that each agent has to carry. As a result, essential parts of the robot control software such as computer vision, motion or behavior control now require other testing methods in order to ensure and evaluate the module's quality.

The aim of this thesis is to simplify the development process of a robotic agent's complex control software by guaranteeing reproducibility and predictability for test cases. It is impossible to ensure these two factors in normal test sessions using robotic hardware, even under laboratory conditions. The simulation package also offers each developer unlimited access to the latest state of the robot.

The following chapters provide a brief explanation of how this simulation software has been implemented and all design and concept decisions are presented in detail. To this end, an introduction is given and the need for this work, as well as its general contribution, are explained in *Chapter one*.

In *Chapter two* important terms and definitions are introduced in order to help the reader understand the key issues of this thesis. Furthermore, this chapter includes related work such as alternative simulation solutions by other RoboCup teams, as well as presenting non-RoboCup related challenges solved through the use of simulation processes. The RoboCup initiative, with a focus on the soccer competitions is then briefly described. In order to apply physical forces to a virtual robot model a dynamic package is necessary and thus a set of widely-used physical libraries is specified, and their strengths and weaknesses are compared. For clarification, the

most important interfaces of the FUmanoid robot control software are described in the final section.

Chapter three gives an overview of the most important testing methods using robotic hardware, as it is important to know how the testing processes performed before generated sensor data was used. The test result evaluation process is also explained in this section, with many requirements for this thesis being derived directly from the deficiencies found in these tests.

In *Chapter four*, the idea and the concept of the architecture are briefly explained and the implementation of important modules is presented in detail, focusing on simple integration into the FUmanoid software eco-system and modeling of humanoid robots within the simulation. A closer look at the emulated sensors, such as camera, actuators, IMUs and pressure sensors is also taken and each emulation approach is briefly described, as are all of the manipulation properties.

The simulation software has already been used in the main development cycle of the FUmanoids as well as being used as a sensor feedback generator for two student software project courses. In *Chapter five* an evaluation is presented to determine whether the results of the courses have been transferable into the main project with experiments showing the possible impact of the simulation software in its current state on the development process of three main parts of the control software (vision, motion and behavior control).

An outlook and a conclusion are given in the *final chapter*. Ideas for further improvements are presented along with changes required to make the development process using the simulation more efficient and useful for the developers.

1.1 Contribution of this Thesis

In recent years many different software solutions have been developed by other RoboCup teams providing emulated sensor data and simulating physical forces for the dynamic processes of the virtual robot structure.

The testing scenarios of teams participating in RoboCup Soccer are often very similar and the modeling structures of the different robot control software solutions are likely to have a lot in common. The simulation software therefore has a clear interface to its sensor outputs allowing the data packages to be easily adapted into every control software.

In the case of physical feedback there should be no dependency on any dynamic library; thus this thesis provides access to a wide range of different packages. Depending on the use case, the library can be exchanged, or its suitability as a physics engine in robotic soccer compared to other packages can be evaluated with this software.

Furthermore, this thesis introduces a new data structure in order to describe humanoid robots in a definite manner. The Robot Description File is now used in other FUmanoid software modules or plug-ins and can be seen as a centralized file containing all relevant information about each FUmanoid robot generation.

2 Background and Related Work

This chapter defines important terms and introduces and discusses the key issues of this thesis. Furthermore, related work is presented and the need for this work is motivated. An introduction is given to the RoboCup initiative and the Humanoid League competitions. Examples of existing software packages for FUmanoids that directly interact with the simulation software are then briefly described.

2.1 Simulation

The simulation and emulation of robotic systems and their sensing devices is a very important part of the development process of robotic hardware and software. The terms *simulation* and *emulation* in the context of robotics have been defined and distinguished very differently in literature. Before 1980, *emulation* referred only to the emulation with a hardware assist, but currently the term often means the complete imitation of a machine executing binary code [40].

In contrast to this, the term *simulation* is defined by the Encyclopædia Britannica as the use of a computer to represent the dynamic responses of one system by the behavior of another system modeled after it. When the simulating program is run, the results will be an analogous to the behavior of the real system[3]. The output data can be used by other programs without any further step of adaption. Simulations are especially useful in enabling observers to measure and predict how the functionality of an entire system can be affected by altering individual components within that system. Additionally, this means that in the prototyping phase many tests can be run on simulated models instead of building an expensive hardware construction for each iteration.

Compared to real life test scenarios, sessions based on simulated data are independent of time and physical conditions. Furthermore, artificially created data can be used to make predictions about the future. This is a very powerful simulated

data-method, since no real data exists for use cases of this kind. For example, mathematical models are used to simulate weather conditions in the near future.

2.2 Related Work

Simulation is used to solve a variety of tasks in different industries and various fields of application. The expectations on a simulation software vary greatly as does the need for physical or geometrical accuracy. This chapter gives an overview of the different areas of application for simulation, and related projects that have created multi-body system simulators are presented.

In general one can distinguish between three types of simulation solutions

- Planning Simulation
- Multi-Body and Sensor Simulation
- Behavior Simulation

Two examples are given where simulation is an important part in the development process of the final product. Simulation has become a method for increasing the performance of certain processes while at the same time reducing their costs and risks. Further information is also gained by the controlling personnel at a very early stage of the project where no real data would exist without a simulation solution.

Firstly, the influence of simulation tools in the process of the optimization steps in the design of a racing circuit is presented. In the last decade, driving simulators have become an important tool for studying driver performance under different traffic and environmental conditions, and it has also become reasonable to use these powerful software solutions as tools for assisting geometrical road design [7]. Furthermore, Benedetto describes the progress that this kind of software has made within the last few years and presents the first results of a driving simulation based study to analyze engineering problems in the design phase of a Formula 1 circuit as well as on single circuit improvements. Benedetto et al. state that the optimization of a circuit largely depends on two factors [7]:

- physical: structural features such as the impact speed in the case of a run off accident
- geometrical: perceptive features such as the influence of lane dimension on driving performances

Secondly, in a totally different scenario, Airbus used several simulation models in

the planning phase of its new A380 airplane and for training programs of airline employees such as pilots and technicians.

During the aircraft's concept phase, Airbus cooperated with Airport Research Center GmbH in order to create simulation models for general ground handling and boarding and disembarking processes [5]. As a reason why a simulation was chosen over a test session in reality, the company stated that tests with real equipment need huge resources in terms of staff, equipment, coordination, and time, and would tend to detect problems too late. As a matter of course, the dimension of the wide-bodied Airbus A380 with its new cabin layout also makes great demands on the boarding and disembarking processes. Therefore, the simulation was used to analyze different cabin layouts and inspect the boarding processes within this new cabin type [5].

In order to prepare future personnel for their work on an Airbus A380, Airbus's Maintenance Training Center in Hamburg introduced two virtual reality devices called MFTD (Maintenance/Flight Training Device) expressly for the purpose of maintenance training [4]. The Virtual Aircraft simulates the complete airplane and enables trainees to walk around the aircraft and visit different work areas. Furthermore, it is possible to work on single components, including removing and replacing them in a three-dimensional on-screen environment. As a result, Airbus states that some 50 percent less time needs to be spent on the actual aircraft due to the virtual reality system. In addition, trainee feedback showed that they are happier with the clearer understanding they have from the new program [6].

Similar to the preparation of technicians, pilots need simulator-based test sessions before actually flying the airplane for the first time. Pilots with a license for an Airbus A340 were allowed to take part in retraining for the A380. Four weeks of theory and around 30 hours in the simulator were necessary before a pilot was allowed to have an initial training flight in the real airplane.

Multi-Body System Simulators

Currently there are many simulation packages that support the simulation of multi-body systems that have been developed in the past years. They can be categorized into three different groups, rarely targeting the same group of developers or applications.

- commercial simulation packages
- free open source solutions

- individually built systems to solve very specific problems or to integrate into an unconventional software suit, such as handling the robotic software eco-system of a RoboCup participating team.

In the case of a student project like the FUmanoids, a commercial solution is too expensive and often many interesting parts of the software are proprietary and closed source, which means that they can not be modified.

Chapter 2.4.1 introduces dynamic package, which are used for collision detection and model physical reactions of the colliding bodies. Usually, each suite has a fixed integrated dynamic package, generally ODE in the context of RoboCup.

A closer look is taken at the other simulation packages in the following section. The developers focus and the software's chosen field of application will be briefly described.

SimRobot With the addition of support for the NAO humanoid robot, by the french company Aldebaran, *SimRobot* [26] now supports three classes of robots including vehicles and four legged robots. The simulation case is defined by an XML-based description language, called *Robot Simulation Markup Language* (RoSiML) [20], developed by researchers from the University of Bremen and the Fraunhofer Institute for Autonomous Intelligent Systems. ODE was chosen for the rigid body dynamics. Qt and OpenGL were used for visualizing the 2D and 3D representations respectively. Each object in the world is part of a giant simulation tree and can be manipulated through the GUI's by a wide range of views [25]. Furthermore, SimRobot supports different sensor types, such as cameras and servo motors, which are well integrated and can be plugged into a certain physical object. The SimRobot source code is freely available under GPL license. It is updated every year and released by the BHuman team from the University of Bremen [36].

MuRoSimF Another individual software solution is the *Multi-Robot-Simulation-Framework (MuRoSimF)*, developed by Dr. Martin Friedmann from TU Darmstadt, Germany. This simulation framework was presented in papers at the RoboCup Symposium 2007 in Atlanta [18]. Instead of utilizing existing dynamic packages, the MuRoSimF has its own algorithms for the simulation of dynamic processes. This is so that the algorithm can be easily exchanged depending on the simulation use case, e.g. when less accuracy is needed a faster solution can enable the MuRoSimF to evaluate more test runs in less time or handle more simulation participants within the same scene [17, 16]. The 3d

visualization is also done in OpenGL. Currently, the source code of this work has not been published.

USARSim A software solution for Urban Search And Rescue simulation (USARSim - today it stands for *Unified System for Autonomous Robot Simulation*), was developed by three cooperating research institutions in the USA [2]. Although the University of Pittsburg, University of California and National Institute of Standards and Technology designed the simulator originally to support rescue applications with robots, it can also be used as a general purpose multi-robot simulator by extending it to model arbitrary application scenarios. As a dynamic package and visualization framework, USARSim uses the well-known Unreal-Engine 2.0 [13]. Via a TCP/IP connection between the simulation framework and the control software with which the simulated data is exchanged. The source code is available for free under the GPL license, excluding the files using the Unreal-Engine.

Webots As a representative of the group of commercially available robot simulation packages, Webots offers a wide range of robot classes, including vehicles and legged robots. ODE is the chosen physics engine and for visualization OpenGL is used. Furthermore, Webots provides robot models such as Sony Aibo, Khepera, or Pioneer2 by default. The scenery is described using the *Virtual Reality Modeling Language (VRML6)* [1] with special nodes to add all the information for a robot and its body parts. Data is exchanged between control software and Webots via a TCP connection. It offers programming APIs not only for C or C++, but also for Java, Python and MATLAB [28].

As shown below in Table 1, all simulation packages released as open source use only one exchangeable simulation method for the dynamics. Exchanging the dynamic package is complicated, because many components within a simulator software depend directly on the the dynamic library, and the architecture was not designed for such a use case.

Simulation Package	Dynamics Library	External Sensors
SimRobot	ODE	yes
Webots	ODE	yes
USARSim	Unreal Engine	yes
MuRoSimF	unique solution	yes

Table 1: Overview of the simulation packages

The emulation of a sensor, like a camera or an actuator, is generally supported by many of the stated packages. There is a big difference, however, in granularity of the virtual sensor configuration, to take it as close as possible to the properties of the devices used in reality.

2.3 RoboCup

Three years before "Deep Blue" - an IBM computer that specialized in playing chess - achieved its great victory against Garry Kasparow, the "Robot J-League" was founded in June 1993 in Tokyo, Japan. This competition was the predecessor of the "Robot World Cup Initiative" - the RoboCup. Right from the outset an ambitious goal was defined, stating that, by 2050 a team of humanoid robots would be able to play and win against the then current FIFA world champions.

Choosing soccer over the chess scenario as the main method for comparing new research achievements in the fields of robotics and artificial intelligence was a considered step, as soccer has a simple set of rules and does not need a lot of equipment to build up a testing environment. Compared to chess, soccer offers a totally dynamic game play, while chess is turn-based and the chess pieces have a fixed set of properties, like a certain chess move behavior. Chess had been very popular in artificial intelligence, but by the end of the last century most of the improvements could be made by increasing cheap memory and processing power instead of making major changes to the algorithms.

The RoboCup soccer competitions are thus just one step on the long road until autonomous robots can enrich our daily life as personal assistants. To this end, new leagues with different areas of focus have been established in recent years, e.g. RoboCup Rescue and RoboCup @Home. In these new scenarios robots are tasked to find and rescue victims in an unknown terrain, find and grab products in a local store or assist a human at home by serving a drink, among other examples. Each RoboCup scenario requires a combination of different required skills, such as cognition, communication, navigation, cooperation and motion control.

Since 1997 annual tournaments have taken place and offer a platform for transparent comparison of recent research results, as well as sharing new ideas with the RoboCup community. To assist with this sharing of ideas a "RoboCup Symposium" is held every year, and selected papers and posters are presented to an international audience.

Figure 1: RoboCup-Logo[a]

Besides talks on the technical challenges, social and ethical questions concerning robots interacting directly with human beings are also on the agenda of the symposium. In 2010 for example, Professor Chong Tow Chong presented his talk *The Three Laws of Robotics and the Coexistence of Human and Robot in Harmony* as a keynote speaker.

2.3.1 Humanoid League

In the first years of RoboCup most of the innovation was driven by the improvements to robots on wheels, i.e. the Small-Size League or Middle-Size League. Due to their higher acceptance by human beings and their advantage in operating environments originally constructed for humans [23], humanoid robots found their way into RoboCup in 2006.

In the first few years competitions with bipedal robots lacked matches with complex behavior, as the robots struggled with their ability to walk stably over the soccer pitch. The challenges were then mainly reduced to simple foot races and penalty kick competitions, before humanoid robots played their first soccer match in 2007.

The humanoid league is divided into three classes, based on the height of the robots:

- KidSize League: 30 - 60 cm
- TeenSize: 100 - 120 cm
- AdultSize League: 130 - 160 cm

The humanoid league has strict rules on the size, shape and color of the robots as well as the game play of a robotic soccer match. Year after year these rules are updated to reflect the FIFA rules [19] and to adjust the robotic soccer field to become more and more like an official pitch.

A very important rule is that robots have to act fully autonomously; it is not allowed to send any information from a central computer. Communication by WiFi between teammates is allowed however to gain global information about the game-state, e.g.

to exchange their current estimation of their own position.

Currently, every object in the environment has been allocated a certain color. The ball, as the most important object, has to be orange. The goal posts are painted in yellow or blue. Extra landmarks, called side-poles, which simplify the orientation problem, are placed at each side of the half-way line and painted in yellow-blue-yellow and blue-yellow-blue combinations. To separate the robotic teams, cyan or magenta colored team-markers must be worn. Fieldlines have to be white and the pitch carpet is always green. Body parts of the robot can have black, white or silver-like colors.

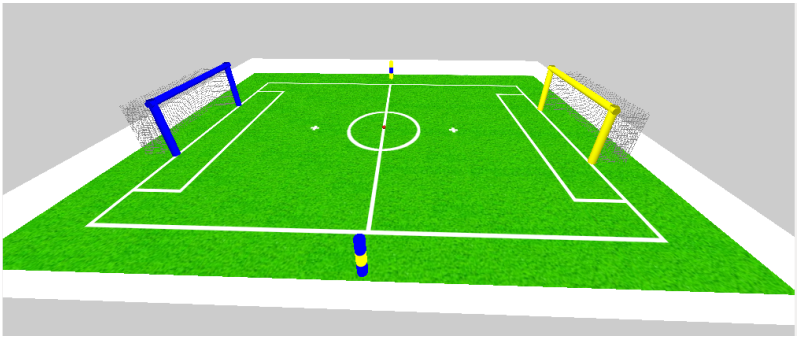


Figure 2: Image of an official RoboCup KidSize League pitch based on the 2011 rules [33]

Humanoid Soccer Competitions

In the case of the KidSize robots, the soccer pitch measures $6 \times 4m$. Since teamwork is an essential part of this competition, every team is allowed to play with three robots at the same time. During the match, robots are faced with many simultaneous challenges. In this competition, computer vision and dynamic walking are the key issues for successful participation. An agent has to identify the objects next to it in order to identify its location on the pitch and decide what it should do next. In 2011, most of the teams that reached the KidSize quarter finals showed a good walking performance with a walking speed above 20cm/s.

The ball is the only object the robot has to interact with. A reliable cognition of the object is therefore very important. The ball has to be tracked and its direction of movement and velocity have to be extracted from the raw data given by the camera sensor.

2.3.2 Team FUmanoids

The FUmanoid Team from Freie Universität Berlin has been participating in the Humanoid KidSize League since 2006. The project is part of Prof. Dr. Raúl Rojas research group and mostly driven by students working on their bachelor, master or diploma theses. In the beginning, a small team of five people and two robots built from a Bioloid Kit [29] successfully participated for the first time at RoboCup 2007 in Atlanta and were placed 3rd. In 2009 and 2010, the team achieved its biggest success by placing second at the World Cup in Graz and Singapore.

RoboCup as a research field for students has a long tradition at the Department of Computer Science of Freie Universität Berlin. The FUmanoids are the successor of the FUFighters - a well known and very successful former RoboCup team participating in the SmallSize and MiddleSize League from 1999 until 2006. In order to fulfill all the requirements for promising participation the team consists of students from different subjects, such as computer science, mechanics and electronic engineering. This diversified grouping allows students to see the bigger picture, as each will explore different aspects of the same problem.

2.4 Simulation in RoboCup Context

In this thesis, a RoboCup-specific physical environment for humanoid robots is simulated. The attached sensors are emulated and the sensor feedback is transferred to the robot control software. This process occurs transparently so that the inner modules of the control software are not aware that they are processing simulated data.

The control software is the most important piece of software within the development process of a robotic agent, regardless of its application and the RoboCup competition. In some cases the software has to control several agents (in SmallSize or Simulation Leagues), but in others each agent has its own instance of the software on-board, e.g. in the MiddleSize, Standard Platform and Humanoid Leagues.

Nevertheless, simulation is highly relevant in both approaches. One problem the developers all share is the challenge of the systems complexity and as a result the hard task of ensuring its quality. A debugging process can be very tough in a software project with many contributors and integration of third party code. Not only are concurrency problems within an control software process an issue, but also

dealing with several threads. This scenario becomes dramatically harder to manage when several instances of the robot control software all cooperate in solving a task by communicating and exchanging information.

Simulation can provide a framework for all kinds of different testing methods. Due to its ability to reproduce a certain test case several times it is ideal for training parameters with a learning algorithms. Furthermore, such a framework can manage automated test scenarios, which are not achievable when real hardware is needed in a test case.

Besides using simulation as a quality metric for the robot's control software, it can also be applied to the testing processes of the iterative hardware development process. Thus, it is possible to test, whether existing motions can still be implemented on the new design after new body parts have been designed.

2.4.1 Widely-Used Dynamic Packages

Depending on the usage of the simulation it may be necessary to deploy a dynamics package, e.g. to generate forces for each mass or to discover if a collision between two masses occurs within the last step of the simulation.

Many packages already exist for this and they are also widely used in RoboCup environments. A major drawback of this however is the problem of a total dependency on the choice of package.

In this chapter a brief overview of the most common packages is given.

Integrators

Each dynamic package has to manage the properties of all physical body objects such as position $p(t)$, velocity $v(t)$ and acceleration $a(t)$. One possible solution to distinguish these values after a given force acted on a body is Euler's method, where

$v(t) = \int a(t)$

$p(t) = \int v(t)$

Adrian Boeing described the method of the Euler Integrator [8] and it is introduced as follows as a common solution for the basic problem of dynamic libraries.

$p(t_0 + h) = p_0 + h\frac{dp}{dt}p(t_0)$

This equation is called Euler's Integration, where starting at some time, t_0, the value of $p(t_0 + h)$ can then be approximated by the value of $p(t_0)$ plus the time step h

multiplied by the slope of the function, which is the derivative of $p(t)$. Furthermore, this is a first-order numerical procedure for solving ordinary differential equations with a given initial value.

By using Newton's second law of motion, the equation can be written in the form

$p(t+h) = p(t) + v(t)h$

$v(t+h) = v(t) + \frac{f}{m}h$

where f is the affecting force and m is the body's mass. Even this small estimation leads to large numerical inaccuracies, unless very small time steps are chosen. A small time step automatically increases the computational effort and the whole approach can become inefficient. Therefore, many different methods have been implemented, which is one reason for the big differences in the accuracy of dynamic packages. This method is an example of an explicit method, which means that the new value $p(t+1)$is defined in terms of values that are already known such as $p(t)$.

In contrast, an implicit integrator such as the Euler's backward method will improve the solution, but has the disadvantages that knowledge about future states of the system is required.

$p(t+h) = p(t) + v(t+h)h$

$v(t+h) = v(t) + a(t+h)h$

The only approach providing future states is to work with approximated states, but this is hard to solve. A common solution to improving the accuracy of the integrator is to increase the order of the estimation to include updates at subintervals of the integration, e.g. using a Taylor series. Typically, a fourth order method is implemented, as it provides robust numerical solutions when combined with an adaptive stepping method.

Euler's Integrator in its first order form is used by the *Open Dynamics Engine* project which is described below.

Open Dynamics Engine

The *Open Dynamics Engine (ODE)* [39] is probably the most common solution in RoboCup as a dynamic package for simulating the behavior of robotic hardware. ODE is primarily designed to be used in interactive or real-time simulation, e.g. in video games. Its strength is in simulating articulated rigid body structures, which means structures out of several rigid bodies connected through joints of various kinds

[39]. This includes legged models, such as in the case of the Humanoid League, where the legs are connected to the body.

The chosen simulation method is based on the velocity model, where the equations of motion are derived from a *LaGrange Multiplier*. The official manual states that only a first order integrator is currently used, which was described in the previous section. Higher order integrators have been announced for the future, which will increase the accuracy of the simulation.

Each rigid-body ODE object has a set of properties, such as mass, center of mass and its distribution. These values are static in the case of a single object, which is always the case in ODE. Other property entries may change over time, such as the rotation of the body in space or its position.

All types of joints necessary to create a virtual robot are supported by ODE, such as hinge, two-hinge and slider joints. Furthermore, ODE offers the possibility of allowing joints to act as motors.

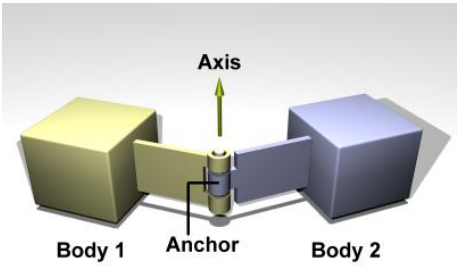


Figure 3: A hinge joint connects two rigid bodies at an anchor point, rotating around a selected axis.[b]

ODE comes with built-in collision detection algorithms. They are not mandatory and can be ignored and replaced with different approaches. So far, the supported collision primitives are sphere, box, cylinder, capsule, plane, ray, and a triangular mesh. Collisions are not evaluated in the normal space of objects, as N-1 tests would be necessary for N geometries, meaning the algorithm would have a run time of $O(N^2)$. Therefore, three types of collision spaces are available: Quad Tree, Hash Space and Simple Space. Within this space collision culling is performed. and this means that pairs of potentially colliding objects can quickly be identified [39]. ODE is operating system-independent and offers a good C/C++ API for its integration.

One drawback, especially in achieving very precise results, is also a built-in feature. The developers emphasized speed and stability over physical accuracy, which makes

sense for their field of application but is one reason why ODE does not represent the ideal stand-alone solution as a dynamic package for bipedal robots.

Bullet Physics Library

The Bullet Physics Library is described as a professional open source collision detection, rigid body and soft body dynamics library in its manual [12].The dynamic package is freely available and also free for commercial use under the ZLib license. This qualifies the package as a serious alternative to commercial closed source projects and is a key reason why it is used in applications for mobile phones or gaming consoles. It can be used on a wide range of platforms, including Playstation 3, XBox 360, Wii, PC, Linux and Android, Mac OSX and iPhone.

The Bullet library offers different collision shapes such as concave and convex meshes and all basic primitives (Box, Sphere, Cone) and allows developers to register their own custom collision detection algorithm, if necessary. For the built-in solutions, the broad phase collision detection is able to sort out pairs of dynamic objects based on axis aligned bounding box (AABB) overlap. Furthermore, it provides a fast and stable rigid body dynamics constraint solver and vehicle dynamics and in the case of joints of the basic types such as slider, hinge and a generic 6DOF are available.

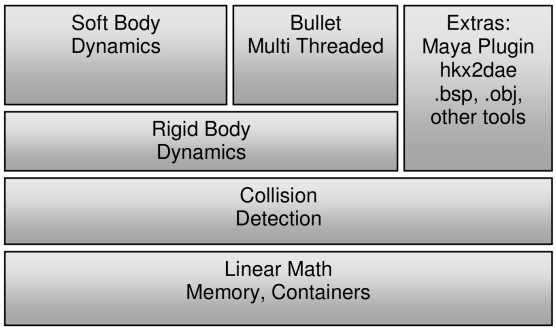


Figure 4: The Bullet Physics Library has a highly modular build [12]

The architecture of this dynamic package is highly modular and keeps dependencies to a minimum. It allows the use of selected parts of the engines such as the collision or rigid body dynamics component. The structure is shown in Figure 4. The library supports multithreaded execution and the use of several CPUs and GPUs as well as 32 and 64 bit systems. To increase the accuracy of the simulation results, it is possible to compile the library as a double precision version, which leads to a lower performance of the system.

Nvidia PhysX

PhysX, developed by NVidia, is a very powerful dynamics package and well distributed in the video game and movie industry[30]. It is freely available as a SDK for non-commercial use, but the project is not open source. It has rarely been chosen as the default library in Humanoid League so far. Like ODE, PhysX offers versions for Windows, OSX and Linux in 32- and 64-bit. It also allows multi-threading on CPUs and NVIDIA graphics hardware.

As well as supporting the physical simulation of rigid body objects, soft bodies, volumetric fluids, cloth and clothing can also be simulated. The last two categories may not be useful in RoboCup, but they demonstrate the wide variety of supported materials, e.g. clothing requires further features such as a self-collision detection system [30].

When simulating the robots hardware, these extras can be completely discarded [30]. In addition, PhysX is also able to group collision participating objects, filter collisions and disable or enable the collision behavior for specific objects. Also a continuous collision detection system is available as well for fast moving objects. The collision primitives are sphere, box, capsule, plane, heightfield, convex, and triangle mesh.

Nvidia PhysX performed the best in the Axel Seugling and Martin Rölin detailed engine comparison [38].

2.4.2 Approved Sensors in Humanoid Soccer

As described above, the Humanoid League restricts the use of sensors to passive sensing devices only. This includes cameras, inertial measurement units, as well as pressure and acoustic sensors. According to the RoboCup rules a sensor has to be human-like to be accepted. This means that a human organ that fulfills the same task as the sensor itself has to exist, e.g. a camera system and the eyes, both used for visual recognition.

The camera is the main sensor for a humanoid robot in exploring its environment. In 2011 a robot was allowed to carry up to two cameras as long as both of them were placed in the head. The total field of view is restricted to less than 180 degrees, regardless of whether one or two cameras are used.

In comparison with robots on wheels, bipedal robots are faced with the possibility of falling down and walking slowly and unstably, due to inaccurate or missing ground

contact information. Therefore, binary switches or more complex pressure sensors must be used. It is also possible to measure the center of mass, which is always in motion when a non-static walking algorithm is applied.

In the Humanoid League the size, weight, robustness and costs of a sensor have a significant impact on further design decisions. Due to this low cost sensors are often chosen to decrease the total cost of a humanoid robot platform, providing their performance is still acceptable. For this kind of robot class the actuators are the main unavoidable expense. The total number of degrees of freedom cannot be dramatically reduced and cheaper servo motors often suffer from less power, speed and precision.

As a result, sensor fusion is a powerful way to increase total accuracy by combining several sensor measurements. For example, a set of four pressure sensors per foot, an IMU in the body of the robot and the current position of each actuator in a joint can be combined to give a precise position and rotation information for the camera in the head. In this way, each piece of information is used to correct errors in the measurement of another sensor. This can be done using a Kalman Filter for example.

It is however still necessary to have an accurate timestamp for each data set generated by one of the sensors used in the system. In a self-made humanoid robot data is usually collected on a microprocessor board through different input ports (e.g. USB, serial connection, Firewire). It is important to emphasize that all data sources have different behavior in terms of latency and the resulting round trip time for data packages, which must be taken into account at other points of the system.

2.5 FUmanoid Robotic Platform

The FUmanoid platform is defined as everything that is necessary for an agent to exist and solve tasks autonomously, including its hardware, electronics and software. In this chapter, important parts that are related to the simulation software will be introduced. Firstly, the robot generations are presented and their differences are emphasized. Secondly, the robot control software of the FUmanoids is described with a focus on the interfaces, which became important for the development of the simulation software.

2.5.1 FUmanoid Hardware

All FUmanoid robots within one generation are identically constructed, so that each robot is able to fulfill all tasks in a soccer game. Furthermore, a robot is fully replaceable by another one in case of damage or electrical problems. The different robot generations used within the last 2 years differ mostly in their specific leg construction and the electronic hardware devices used. Relating to the simulation software of this thesis it was necessary to represent highly diverse robot construction within the same software package.

Both FUmanoid robot generations will be briefly described as follows.

2009 Robot Generation

The robot generation developed in 2009 involved a complete redesign of the former robotic platform, with the plans and majority of construction being conducted by the student Mariusz Kukulski as part of his diploma thesis [24].

One key feature of this construction is its double joint knee, which can be bent in both directions in the same way. This offers the possibility to walk backwards and forwards with the same dynamic walking approach.

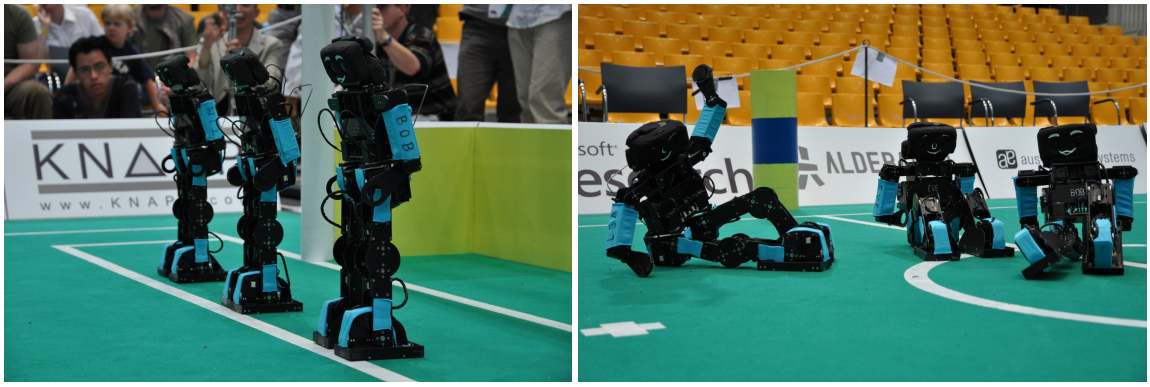


Figure 5: The FUmanoid robots in Graz, Austria (2009)

A major drawback of the former platform was the lack of computing power. Therefore, in 2009 the Gumstix Verdex Pro XL6P [14] with 600Mhz was chosen as the new main processor.

A stereo camera system designed by Bennet Fischer unified all important processing and sensing devices into one board, so equivalents to eyes and the brain were all stored in the head of the robot [15]. This approach was canceled in a later iteration

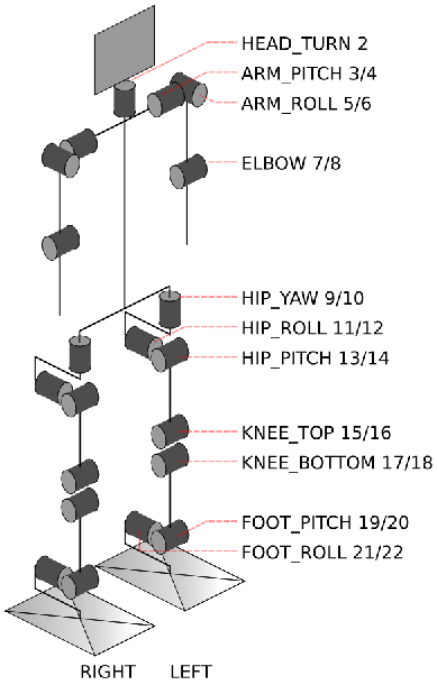


Figure 6: A schematic structure of a robot.[c]

of the robots design which reverted to a single camera setup resulting theoretically in a doubled frame rate.

An IMU with five axes and four binary switches per foot provides information about the body posture of the robot. Each switch indicates ground contact of the related part of the foot. This data is very important for walking stabilization and for a precise calculation of the camera position and orientation.

2011 Robot Generation

In this section all elementary changes made to the 2009 version of the FUmanoid robots are described. The redesign was based on the analyzed data and the experience from 2 years and several competitions with the former model and the changes which mainly affected body parts and electronic hardware. The control software was adapted through smaller changes, as described in the following section 2.5.2.

A major modification was the replacement of the main processor by an OMAP3 processor with 1 GHz on a IGEPv2 board, offering increased performance compared to the former Gumstix processor. The self-constructed board including the stereo camera system was not robust enough and a common cause of failure in the robots.

Therefore, it was replaced by a Logitech Quickcam webcam, which delivered images in a better quality for further processing steps.

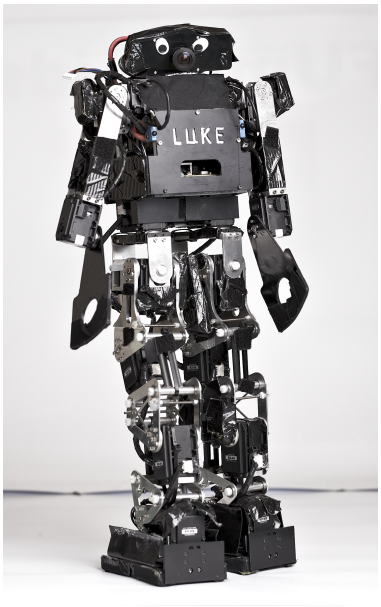


Figure 7: Prototype 'Luke' of the 2011 FUmanoid robot generation.[d]

The legs were constructed in a completely different fashion compared to the old ones. Based on the concept of parallel kinematics, the double joint knee was exchanged with a fixed construction, which results in the foot, knee and hip layer always being positioned parallel to each other as shown in Fig.8.

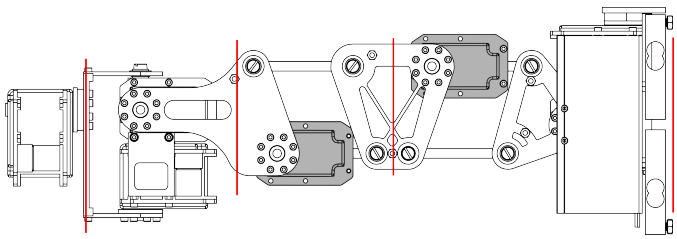


Figure 8: Image showing the concept of parallel kinematics. Each layer (red) is always in parallel to the others. [e]

For more precise information about the ground contact of a foot, the old foot switches

were replaced by a load cell, normally used in kitchen scales. Four of them were plugged into each end of each foot to measure the exact weight on that particular point. The idea is to continue to obtain the center of mass information as long as at least one cell has contact with the ground.

Figure 7 shows "Luke" one of the current models of the FUmanoids.

2.5.2 FUmanoid Control Software

Robot control software needs to clearly differentiate certain layers. On the one hand there is high level control, which is used to decide the upcoming actions. And on the other hand, low level control takes care of applying these actions within the robotic hardware itself, e.g. by controlling actuator positions to give the appearance of stable dynamic walking.

Control mechanisms can be divided into two groups, the *Reactive Systems* and *Deliberative Systems*. The first group manages sensor inputs and reacts directly with a decision based on a combination of the states of the given inputs. A deliberative system on the other hand manages a data structure, which can be seen as storing a brief description of the surrounding environment - a world model. These systems are therefore highly dependent on the accuracy of the given model. For the successful application of a deliberative system, information within the world model has to be stable and the rate of change of every entry has to be small in order to plan ahead.

Although both approaches seem to be very different, both follow the structure of *sense*, *plan* and *act*, which is a central concept in software design for robotic systems [11].

Sense In this phase the robot collects information about the environment or its own state using its sensors.

Plan The *Plan* phase uses the information from the *Sense* phase in order to decide what should be done next. For example, calculating the shortest path to the ball.

Act As the final step, the control software executes these planned procedures in the *Act* phase.

In most designs involving a full sense-plan-act cycle, a set of different modules have to interact with each other within a robot control software. Furthermore, this set of modules looks different each time the robot is used in a different application, while the concept of sense-plan-act can still be applied.

Any future mention of the FUmanoids robot control software within this work does not refer to its initial architecture design by Daniel Seifert [37] from 2009, but to the RoboCup 2011 release version, which can be seen as a logical, iterative development.

As shown in Fig.9 the *WorldModel* module is the core data structure of the architecture. It stores the sensed data and offers each piece of information to other modules. The Behavior module uses current world model data to make further decisions, which are then passed in to the walking module in order to move the robot to a specific position. Sensors (marked gray) provide input, which is interpreted by modules, such as *Vision*. The *WorldModel* data is also shared with every other robot belonging to the same team. In order to create a clear and easy understandable interface the communication protocol has therefore been changed to *Googles Protocol Buffers* [21].

The relevant modules and interfaces for this work, are as follows:

Network The communication module provides the interface with the outside world. It dispatches incoming data packages to the correct recipient or reacts directly to certain messages.

Actuators The interface with the servo motors is called *MotorBus*. It handles read and write requests for each component on the bus. In a later development stage, other hardware devices are added to the bus, like the IMU or the load-cells in the foot.

Computer Vision As the entity for cognition, the task of the vision system is to identify objects, such as field lines, goals and the ball, in an image provided by a camera.

Localization This module uses vision data to position the robot on the soccer pitch.

WorldModel As the central data structure, the *WorldModel* stores all extracted information. As well as vision and localization information, team and game state data are continuously evaluated from received network data packages.

Behavior This module uses WorldModel data to makes further decisions such as the robot's current walking path or general teamplay actions.

Many modules, especially the ones handling open network or device connections, need to run constantly. In order to achieve this, the FUmanoid control software consists of several threads running concurrently. While the main processor is a single core, this does not result in truly concurrent results in a multiprocessing system.

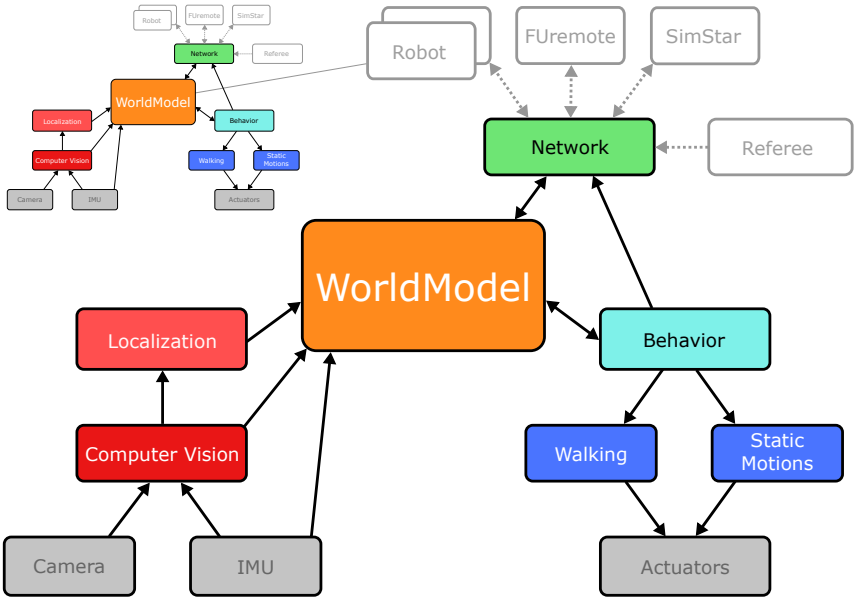


Figure 9: Class diagram showing the internal relations of FUmanoid control software

The control software interacts with any other tool in the FUmanoid software ecosystem through UDP or TCP network communication. Communication paths that have to exist during a match use UDP connections, because the official rules forbid a soccer match with an existing TCP connection. It also offers access to data streams with different levels of abstraction, which was later used by the simulation to transfer simulated data into the corresponding modules.

Network packets containing status messages are exchanged to propagate global information about the state of play. The most relevant status message for this work is *StatusPacket*, which is used to exchange data between more than one robot control software instance. Within a *StatusPacket* message, the robot control software writes all the information it wants to share with another robot or with analyzing tools. In the case of a simulation software, this means that ground truth data can be directly compared to the calculations of a control software module.

3 Multi-Level Hardware Testing

In past years the FUmanoid development team had set up tests to identify system failures and measure the level of performance. All of these tests required the latest robotic platform to be present and in a usable state. However, this method could leave a developer faced with uncertainty. Firstly, the test results were based on observations by the testing team and these observations could be wrong or incomplete. In such cases the conclusion might not just fail to solve the problem, they could even make it worse. Secondly, repeatability of the same scenario is impossible. A robot could be repeatedly faced with an identical task and its environment could be restored as precisely as possible at the beginning of each attempt, but in terms of sensor data and output results from the executed modules, each test set-up would produce different results. Thirdly, one module could always whitewash errors in the behavior of another one.

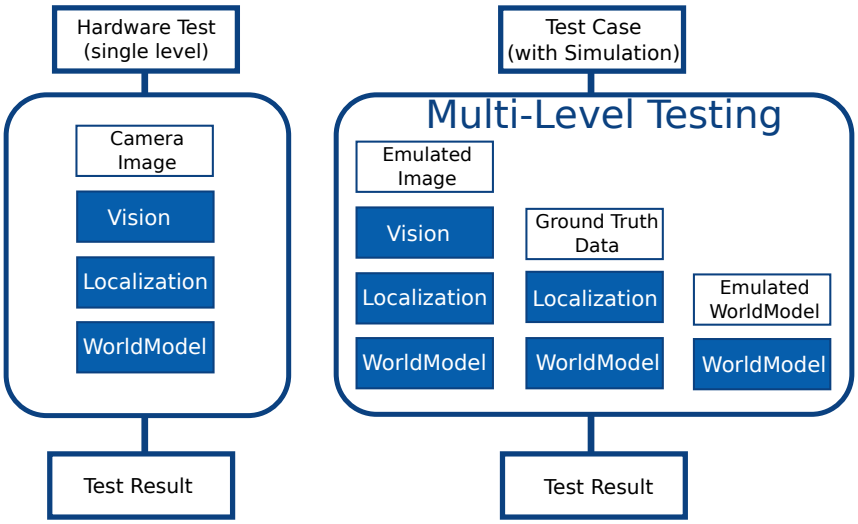


Figure 10: The concept of multi-level testing

Furthermore, developers struggled to create a useful test case after having applied changes to the software, especially for rarely occurring errors. In this case it was

sometimes impossible to check errors as resolved, as there had been no tangible confirmation of the solution.

As well as testing the whole system as a black box, it is always necessary to create module tests, in which each module is isolated from the main system. In this chapter the methods of testing before introducing a simulation software are briefly described. The requirements for this thesis were derived from the former state of hardware tests and its challenges.

3.1 Hardware Test Cases

Hardware based module testing is one way to improve the accuracy of the testing results. During these tests, each module is observed separately without having any dependency on any other module. For example, while testing an localization algorithm the results of object recognition in the vision system have to be correct, otherwise a correctly working algorithm could state a wrong position, due to a failure in a different module.

In order to avoid this, a module test itself needs to be well prepared. Inputs and other modules, which could have an effect on the output results have to be chosen artificially or prepared beforehand to avoid any further uncertainty in the output. A test case for a certain module should be in a stable and ratable state at its beginning. Its complexity should be reduced to a minimal set of requirements that will affect the module. Everything else can be neglected.

Sometimes it is impossible to ensure this when testing a robotic hardware module, because in such a test a robot needs to evaluate real world information measured by its sensors to interact directly with its environment. In this case the relevant inputs for a certain module cannot be manipulated online by the testing team. Therefore, all inputs have to be logged during a testing session. Afterwards, they can be manipulated to ensure the correctness of the input itself and applied again to the module. Using logfiles is a very effective method to improve the quality of the system, but there is also a need to re-analyze and test the software. In 2009, the FUmanoids did not have such tools available, which meant that module tests with the robotic hardware were the most frequently used testing method for the robot control software presented in the previous Chapter 2.5.2.

In the Humanoid League, a robot control software has four major components which can be tested by this method: Vision, Localization, Motion Control and Behavior.

Methods for testing these modules using the current hardware are presented in the following sections.

3.1.1 Vision

The vision system is one of the most important parts of a humanoid robot control software. By analyzing camera images, the robot can explore its environment and the current state of the game, in the case of a RoboCup soccer match. The extracted information is a major part of the robots knowledge base, which itself is the foundation of further decision processes.

In Fig. 11 the FUmanoid computing process is shown. First, a raw image from a camera is received by the control software. A look-up table (LUT) is used to associate the wide range of colors to a certain class (black, white, blue, yellow, red, green, cyan, magenta or unknown). The simplified version of the image is now examined by several extraction algorithms, for identifying objects within the image. Once the objects have been found, their relative position to the robot has to be calculated by projecting the image coordinates onto the field layer.

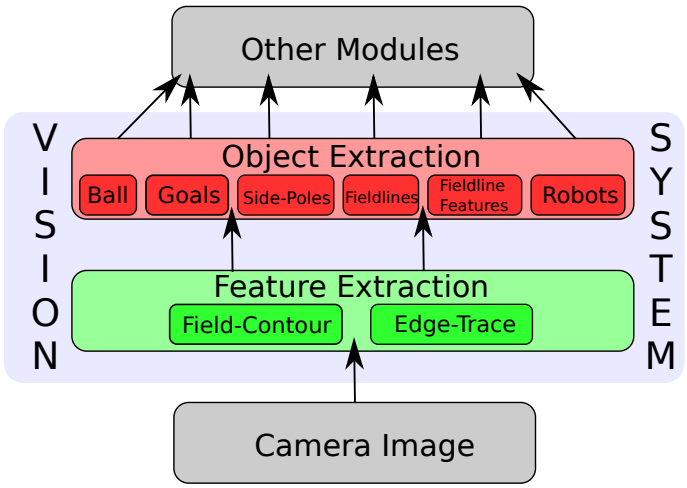


Figure 11: This diagram shows the vision module's data stream

Besides extracting the correct information out of an image, a vision system needs to be able to tolerate small differences in the lighting conditions and preferably also work with various camera systems and lens distortions. In the RoboCup soccer competition the lighting requirements have been loosened over the years, which means that illumination isn't strictly defined by the rules. There might also be an

influence on the images by true light, if the windows of the venue have not been sealed.

Additionally internal changes can have a big impact on the vision system. This includes modifications such as changing the camera lens to a fish-eye type, as previously mentioned: objects will have a different shape and size in the image due to the pronounced distortion of a wide-angle lens.

Overall, the accuracy of a vision system can be described as the ratio between correctly recognized objects within a single frame and wrong or unclassified objects. Because of that, a database of images needs additional meta-data, e.g. which objects can be found in the image at a certain position or region. This kind of knowledge has to be added to the database manually to ensure that it is correct.

3.1.1.1 Validating a Vision System

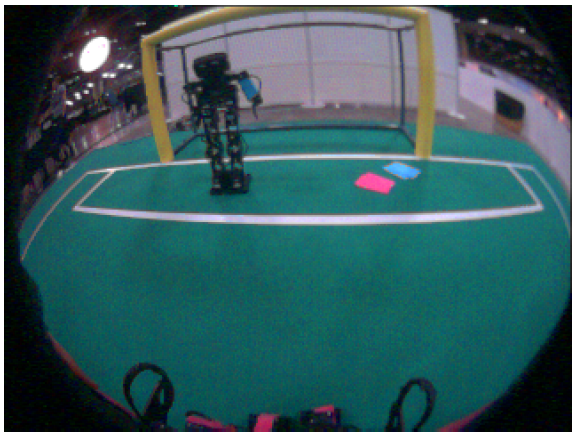
In the past two years an image database has been set up and used to verify the latest version of the vision system or evaluate the best algorithm for a vision-related task such as finding an orange tennis ball in an image.

Image Database

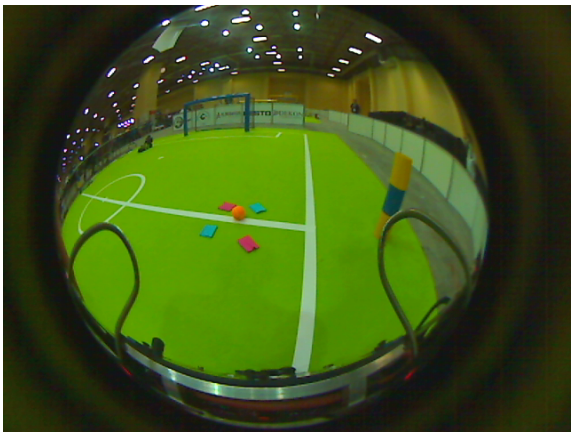
After an image is captured by the robot, it is transferred to a remote computer along with the current color calibration. Additional information, such as the date, time, location and field number is added by the developer on site. Later on, objects are marked and labeled manually in order to provide an ideal case. Every image is then stored in a database, which can then be used for validation processes.

Two types of images exist in this database. The first type of image is taken by a robot at a competition venue, e.g. during a match or in the calibration phase just before kick-off. The advantage of these images is that they were created under competition conditions and therefore include factors such as lighting and distortion - like people in the audience wearing a wide variety of colors. This type tends to have artifacts such as several balls on the pitch or people walking on the field barefoot. These things can unnecessarily complicate the task. In the other type of image the robot is placed in a robotic laboratory in an artificial game state, which was set up explicitly to produce a certain image. This approach suffers from a lack of colorfastness, because RoboCup rules do not state the exact color code for an object on the humanoid league soccer pitch. In other words, objects may not have

the correct colors, and the environment may not be comparable to the one at a RoboCup venue.



(a) 2009: The image is used efficiently and the safeguard blocks information just in the lower part of the image



(b) 2011: A smaller circle with real image information (larger black area) and the safeguard obstructed many more pixels

Figure 12: Two images out of the database

Creating and maintaining such a large database is challenging, because a humanoid robot is a complex hardware system with many properties that can change within a short period of time. For example, in 2010 a new safeguard in case of a robot falling was installed. It was made from heavy wire and painted black like the former protection. Compared to its predecessor this construction obstructed many more pixels in the image, and due to this it was classified as an obstacle by the vision system. An obstacle virtually forces the robot to always walk backwards in order to avoid a collision. This problem had to be fixed by improving the robustness of the obstacle detection algorithm, it was not possible to restore the old behavior with this new hardware modification. As well as requiring adjustments to the vision algorithms, these kind of changes have an enormous effect on the image database for testing. First of all, the new images are not usable for older versions of the vision system. And secondly, older images might not meet some essential requirements of the current hardware setup.

Consequently, there were only a small set of pictures in the database, which were actually used in the debugging process. Considering that these images were often taken at random, there was a huge potential to improve this debugging cycle through the use of a camera sensor emulator.

3.1.2 Localization

By evaluating a single image taken by the robot's camera, it is possible to determine the relative positions of certain objects shown in the image. In order to merge the information from different images, while the robot is in motion, it is necessary to create a global map and then convert these relative positions into global coordinates. This can only be done if the current global position of the robot itself is known. To achieve this a localization algorithm is necessary. A distinction is made between three different basic approaches.

Local Localization In this approach the initial position of the robot is known. The robot's own movement is also required to be tracked. Correct and precise pedometry is used to directly calculate the current global position of the robot.

Global Localization Here, the initial position of the agent is unknown. This means the former approach cannot be applied to correctly solve this problem due to problems with symmetries or converse conclusions, several hypothesis have to be tracked simultaneously. This is more difficult to manage than the local approach presented previously.

Kidnapped Robot A more general approach is the so called *Kidnapped Robot* scenario. In this case the robot might be picked up and placed randomly at a new position, without any feedback apart from the images given by the camera. In the soccer context this moment of total uncertainty in its own position could occur when the robot falls over or after a manual relocation by a robot handler. Even without the problem of unexpected re-locations, this kind of algorithm is very helpful to avoid situations where the algorithm gets stuck and the robot is caught in the wrong position.

The localization method for the FUmanoids is based on a *Augmented Monte-Carlo-Localization (MCL)* algorithm. In this particular MCL three-dimensional particles are used to represent position and orientation on the soccer field[41]. The main source of information is the vision system. In a previous step the vision system extracts features like goalposts, side-poles and field lines, which can be used as landmarks, since their position on the real soccer field is well-defined.

In the case of field lines however, further calculations are required to use them as landmarks. Field line features are not unique, except the circle in the middle of the pitch, and therefore a direct mapping to a specific position with global coordinates is not possible. Three different types of field line features exist on a soccer field. Firstly, an L-feature consisting of two field lines with a boundary point at one end.

Second, the T-feature which also consists of two field lines, but with just one line that ends at the boundary point. The third possible construct is the X-feature, which consists of two lines and one intersection. Each of these possible field line features can be found on a soccer pitch multiple times; to reach a definite state mapping of all identified field line landmarks is performed in order to find a unique position if possible.

3.1.2.1 Validating a Global Localization Algorithm

It is very easy to validate whether the error between the global position calculated by the robot and the actual position on the pitch is small enough to be neglected when used for things such as navigation decisions. By assuming a large error value however it is difficult to ascertain how the algorithm finally selected this particle as its current best.

Two test scenarios requiring the latest version of the robotic hardware have been developed in the last few years, they are detailed as follows:

Overhead Camera System This setup is reminiscent of the SmallSize League. A camera is installed in the ceiling to observe one half of the pitch. Robots are marked with color markers on top of their heads, like SmallSize robots, and the tracking is totally identical. Additionally, a second camera can be added to this setup to observe the whole pitch or a wide-angle fish-eye lens can also be used in a one camera setup. The start position and the exact walking path are the two key pieces of information that can be gained by this approach. Both are used as ground truth data to be compared with the calculated results.

Fixed Path Observation The simplest and most intuitive idea is to determine a specific route beforehand. This could mean a path from one corner of a penalty box to the other one, with an intermediate stop in the middle of the pitch. The robot then has to walk this route and validation is handled by sense of proportion and the calculated position data of the robot.

Both tests can be used as a good criterion as to how accurately the localization algorithm is working. But if an incorrect localization-result is achieved neither indicates what went wrong or, more precisely, which input data lead to the error within the algorithm. As an approach isolating the source of the problem, each step of evaluation in the control software is recorded in a logfile. This method allows the given inputs to be observed frame by frame.

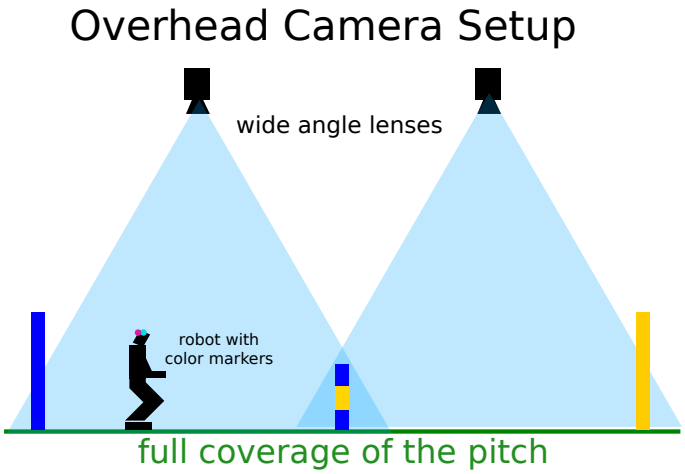


Figure 13: An overhead camera setup with two camera sensors. The robot has color markers on top of its head to simplify the identification process.

3.1.3 Motion Control

As well as reliable image processing, a fast and stable walking algorithm is another highly important requirement in the humanoid league. Without being able to walk, no further tasks are achievable, because this kind of weakness cannot be compensated for.

A robot not only has to walk on its own, but also perform stabilized motions for several other tasks, such as kicking a ball or standing up after falling over. Each motion depends on the kinematic restrictions of the robot's hardware. The more degrees of freedom (DoF) it has, the easier it is for the robot to perform different motions. A high number of DoF also results in improved distribution of the load on each motor. On the other hand, the greater number of actuators means more weight, which in turn makes it harder to stabilize the motion and it also results in higher costs.

There are two kinds of strategies used to handle motion of a humanoid robot.

Static Motions

The simpler and more naive approach is to describe the motion as a fixed order of actuator angle positions and a certain amount of time between each step. At the very beginning the FUmanoid Team predominately used this approach since it made it easy to create and adapt new motions.

There are however many drawbacks to the static motion approach. As a team of

robots consists of a minimum of 3 robots, each motion has to work on all hardware. The smallest modifications to the hardware could lead to instability in performing one or several motions for this robot, which means either that specifically adapted motions have to be created or a new generalized motion that works on every model has to be found. In the same way, static motions might just work on a certain type of surface, for example a thin carpet or hard ground. Changing the location could mean that every motion has to be checked and adapted accordingly. Additionally, the fixed sequence of actuator movements could also be dangerous and result in injury to the developer or damage to another robot; as no sensor feedback is used, a tangle will not be recognized and the motions will not be stopped to prevent damage. The most serious problem though is the fact that static motions by their definition can not react to changing conditions or unexpected forces. For the same reason, walking needs to be handled dynamically in order to take collisions and surface irregularities into account.

While motions to stand up, kick a ball or perform a throw in could still be enacted by the use of static motions, a dynamic solution provides powerful possibilities for more stability and responsiveness.

Dynamic Motions

Contrary to static motions, this concept integrates the idea of a controlled movement. Therefore, feedback is required and up-to-date sensor data is very important. The current FUmanoid robot model takes roll and pitch values from the IMU, load values from actuators and weight measured by each load-cell in the foot.

The strength of dynamic motions is their capability to compensate for minor changes to the design of the robot and the resulting shift of the center of mass by just adapting the motion controller result based on the measured sensor inputs.

Compared to static motions, which can be set up quickly by trial-and-error, dynamic motion control takes more time to develop. Once finished however, it is more versatile and more stable than a comparable static approach.

3.1.3.1 Validating a Motion Control Implementation

Since the very beginning, the test method for motion control algorithms and static motions has remained the same. Motions were simply executed and the quality was evaluated by sense of proportion. While the FUmanoids simply used static

motions for one-time events instead of repeating a motion several times to achieve a continuous movement - such as in a static walking approach - a quality metric was to compare the number of successful attempts to the failed ones. A static motion was considered to be good if the rate of successful attempts was greater than 90%.

In the case of a continuous motion, like a dynamic walking algorithm, it is even harder to find a useful heuristic. First, it is possible to consider the velocity of the robot by letting it run a predefined route and measure the time. Second, stability could be evaluated by counting the number of unstable states during one run.

In combination with a learning approach, the issue with a test run based on a hardware test is that it takes a long time, especially when several thousand attempts are necessary to optimize the learning parameters. This also ignores the possibility that a test run can be interrupted by a technical problem.

In general, a serious drawback is the risk of damage during a test run. To ensure safety, robots have been secured with a cord affixed at the ceiling in order to avoid damage when falling over. Especially when recording static motions using the key frame approach, the robot has to hold a certain pose for a long time. In the worst case, an actuator could break under the heavy load or suffer from a gear wear when a critical pose is attempted too often. Furthermore, the battery status and the temperature of every actuator have a direct influence on the motion itself, regardless of type.

3.1.4 Behavior

Behavior in the FUmanoid context unifies planning, navigation and game strategy. Hence, behavior is very dependent upon the other modules in lower layers, especially when it comes to quality assurance.

As a central data structure, a *Worldmodel* manages and stores all incoming sensor data or pre-processed data in the FUmanoid control software, such as vision inputs or self-localization position values.

Rather than in C++, the behavioral code is written in *The Extensible Agent Behavior Specification Language (XABSL),* which enables developers to describe behaviors for autonomous agents based on hierarchical finite state machines [27].

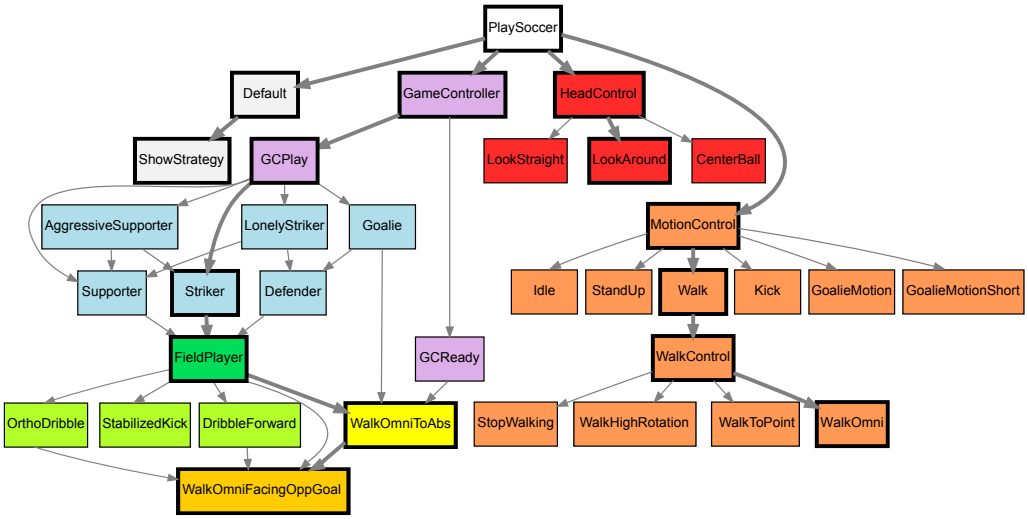


Figure 14: Image showing a XABSL Behavior Graph.[f]

3.1.4.1 Behavior Validation

As previously mentioned, a key issue during a test scenario for behavior evaluation using the robotic hardware is the uncertainty as to whether the seen issue is related to the behavior layer itself or if a problem on a lower layer in the control software caused the planning system to make an obviously wrong decision.

This problem is compounded when concurrent processes are involved, e.g. a 2 against 2 test match where many decisions are based on the merged information of the teammates.

As a partial solution, a log-viewing and analyzing plug-in was added to the robot management software "FUremote" [22] in 2010. This allows a complete replay of all outputs by each agent during the match, which is very useful in detecting the reason for an unwanted reaction. So far this is limited to the robots outputs sent by WiFi only, so no exchange of data between single modules is logged within the FUmanoid control software.

Behavior testing setups suffer most from the inability to reproduce use cases. Each run will be unique and hence very rare and hard-to-find problems might only be seen once. In such situations it is quite likely that this kind of problem cannot be solved at all, since the responsible developer will have only a logfile and the documentation for the problem to work with, without having the chance to reproduce the use case to see it again or to find out whether an applied solution really has fixed the issue.

3.2 Hardware Test Limitations

All possible variations of the aforementioned test methods require the use of the robotic hardware. In this chapter the risks and major problems of this testing method as the sole alternative are examined. In 2009 all quality assurance processes required the availability of the current state of the FUmanoid robotic hardware, moreover multiple agents were necessary to setup one test case.

This method also suffers from the inability to automate the tests and run several thousand tests sequentially with each one having a different set of parameters. Following the application of learning algorithms to solve one of the previously presented tasks, test automation has proven to be useful.

At this point however, all mentioned validation and verification hardware tests were the only possibility to ensure quality.

3.2.1 Test Case Coverage

Many tests based on hardware require a considerable amount of time, meaning only a few attempts can be executed. Not only the test itself, but the preparation beforehand is time-consuming, so ideally only minor changes are made between two runs. In reality even a detailed reconstruction of a test case does not guarantee static conditions for all runs.

In general, hardware tests are very important and definitely indispensable. The limit of the number of test iterations can directly be seen as a restriction of the diversity of possible testing scenarios. This means that even if all tests ended with a positive result, it is likely that an unconsidered test case could have failed.

3.2.2 Hardware Availability

For a RoboCup team such as the FUmanoids, the availability of the hardware is limited and for many module developers access to it is required for further improvements. A FUmanoid robot generation consists of four players. There are usually between 15 and 20 developers working on the project at the same time, which means that the hardware has to be shared or single test sessions have to be postponed.

For many practical tests, especially the ones including network communication, more than one robot is needed by a developer, which further aggravates the situation.

It is rare to have four robots in good shape - without any need for repairs or corrections - at the same time during most of the year. Firstly, the hardware department needs to work on the robot as well, which results in many hardware modifications which could require special software and configuration adjustments. Secondly, in the early stages robot parts tend to break during hardware based tests, resulting in for example electrical problems after heavy impacts.

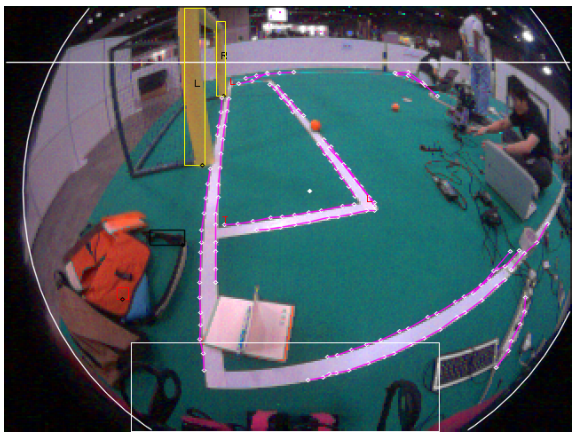


Figure 15: Typical scene during a RoboCup competition. This image shows many unforeseen distortions, e.g. people sitting on the pitch hiding fieldlines or artefacts due to unexpected objects such as the orange backpack on the left, which was extracted as a ball object by the vision system.

Not only the hardware but access to the test environment leads to a clash of interests. While some tests must not be interfered with, such as by other people using the same soccer field, others can not be accomplished at all. For example vision tests usually fail when unforeseen objects are visible in an image, such as blue jeans or red pullovers worn by other testing personnel as in Figure 15

4 Simulation Software

4.1 Software Architecture and Concepts

In this chapter a brief description of the software architecture and design decisions for this thesis are given. Its integration into the existing FUmanoid software environment will be explained in section 4.1.1 and sections 4.1.2, 4.1.3 and 4.1.4 introduce all aspects of the software architecture through a bottom-up approach. The chapter is concluded by surveying detailed information of the visualization methods of data streams within the simulator.

4.1.1 Preservation of Existing Interfaces

As previously described in Chapter 2.5.2, several software packages exist in the FUmanoid ecosystem. Ideally the simulator should replace the complete robot hardware without being identified as an artificial component by the other packages interacting with the virtual hardware agent. Therefore, the design of the simulation software had to suit every existing interface. The exchanged packets between the interfaces can be divided into two groups, *Status Messages* and *Command Messages*. In the following section, the input and output data streams between the main software packages will be briefly described:

FUmanoid (Robot Control Software) The control software sends status messages to inform other instances about the current game state or object positions (e.g. position, direction of movement and speed of the ball). As well as receiving status messages of other robot control software instances, command messages have to be processed (e.g. a robot pose in form of a single key frame for a static motion). All general status messages are sent and received as broadcast messages within a chosen subnet. In the case of a known recipient, e.g. when answering to a request received earlier, the message will be sent as a unicast message.

Game Controller This software program is provided by the Humanoid League to the assistant referee to interact with the participating robots directly. It is freely available and open source in order to ensure the transparent behavior of the referee box. Several game state commands like READY, SET and PLAY are sent during a match from this program. It is also used to call penalties for robots that perform illegal actions on the pitch. The resulting time penalty is further administrated within the program. Such a call forces the robot control software of the penalized agent to stop its interactions until the penalty time is up. All messages are sent and received via broadcast messaging within the chosen subnet.

FUremote (Robot Agent Administration) This project is used in order to visualize the ongoing processes within several robot control software instances. It also includes a plug-in, which reproduces the Game Controller behavior. It therefore receives messages mostly as broadcasts, but always sends explicit messages as unicasts to a known robot control software [22]. As every robot broadcastings its own status frequently, the robots currently running a FUmanoid control software are always known by all FUremote instances listening on the network.

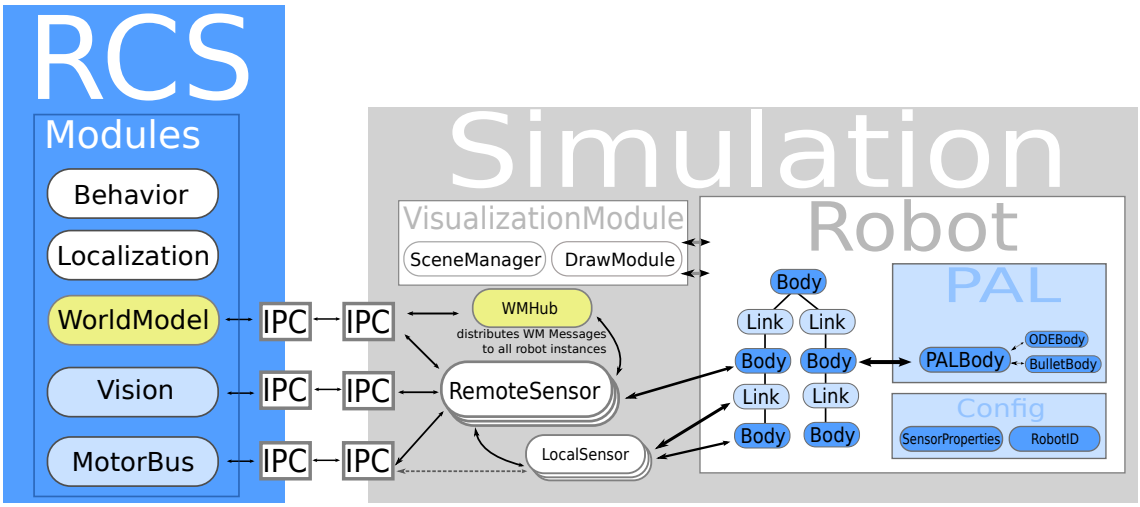


Figure 16: The integration of the simulator is shown in this image

For full integration of a simulation package without abandoning an existing method of communication, an extra layer managing the robot communication was necessary (see Fig. 16). Finally, every program, including all FUremote plug-ins for debugging reasons, should also work with a virtual robot equipped with emulated sensors. The

diagram shows a software environment which is usually run on several machines. The control software is running on the embedded system on the robot's hardware; FUremote is executed directly on the developers laptop and the Game Controller runs on an extra machine controlled by the assistant referee. This setup is not desirable when working with a simulated robot, as generally the developer will want to run all four instances locally on their own machine. Even if this requirement were not to exist, the need to execute many control software instances in parallel in order to simulate a $3vs.3$ robot soccer match made it inevitable that this property was abandoned.

This leads directly to a structural message exchange problem. However, communication via broadcast is required by the RoboCup Humanoid League officials, as it creates a form of transparency and decreases the possibility of cheating. Therefore, the simulation program offers a module, which can be seen as a message hub. This module knows every other participant (all control software and FUremote instances) and duplicates and forwards every received message to all of its ports. This includes messages such as *Game Controller Status, Robot Status* or *FUremote Logging* messages.

4.1.1.1 Robot Description File

Extending the software ecosystem also required the creation of another data structure, which has to make many assumptions about robot models and store many of these as properties. Until now, the FUmanoid project did not have a centralized data structure for such collected data. As well as many general drawbacks during the whole development process, this was a determining factor for several bugs and problems. Firstly, data can be stored distributed and redundant, but is highly likely to become inconsistent over time. Secondly, software modules must include different parameters for specific hardware generations. Code fragments related to a specific robot version are selected by a defined flag at the time of compilation. The program code generally tends to become illegible or at least harder to replicate.

The newly developed *Robot Description File (RDF)* had to fulfill several requirements. As well as solving the problems mentioned above, it had to be easy to access, share, maintain and extend. Therefore, the new format uses *Google's Protocol Buffers* as its main data format. It can store data packages in files and bundle them in a structure which is optimized for transmission via UDP to another network participant. It is also easily extensible and downwardly compatible with former

versions of the created format. Additionally, Google supports many programming languages like C, C++, Java and Python. The Protocol Buffer can therefore be used as an interface between programs written in different languages, such as the FUremote and FUmanoid software.

Having a centralized set of meta-data about all robot generations in the form of a RDF offered an easy way to support each hardware iteration in the simulation package. The robot's model can simply be chosen by the control software before the simulation process starts. Body parts, joints and sensors are generated taking the properties of the chosen RDF into consideration.

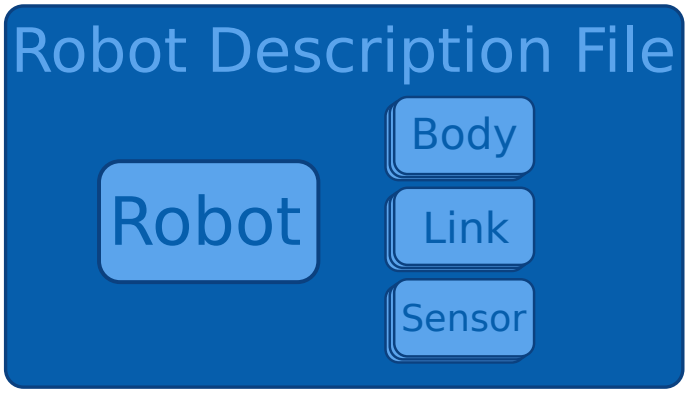


Figure 17: Diagram showing the Robot Description File (RDF) structure

RDF Implementation

Fig. 17 shows a detailed diagram of the current implementation. An RDF describes a *Figure* of a virtual robot consisting of a set of *Body* objects and *Links* between them. Additionally, *Sensor* objects may also be added to certain body parts of a *Figure*. The content of a *Body* entity is shown in Table 2. The position is stored in a four dimensional matrix using a homogeneous coordinate system, including a $3x3$ rotation matrix and a $3D$ position vector placed in the last column. As a quadratic matrix is simpler to process, a fourth row is added containing the identity. A RDF has three primitive body types: box, cylinder and sphere, as well as the more complex body type *BodyCompound*. Multiple primitive bodies are statically linked together building an immutable compound body.

Additionally, *BodyLink* objects are generated out of the data stored in the *Link* properties (See Table 3). With a *BodyLink* object two *Body* instances are connected by a joint, rotating around a given axis. RDF supports revolute, prismatic and

id	int
position	Matrix4d
density	double
type	BodyType
material	Material
name	String

Table 2: Structure of the Body entity.

spherical joints. A motor may also be added to a *BodyLink* structure, if necessary. Therefore, the motor's torque, armature resistance and the back electromotive force (backEMF) can be specified in a RDF.

id	int
parentId	int
childId	int
position	Vector3d
axis	Vector3d
motorId	int
type	LinkType
name	String

Table 3: Attributes of the Link entity.

Concerning the modeling of the environment, a *Material* entity exists to set properties for each *Body,* related to the simulated physical world.

Currently this structure is utilized in the simulation package and motion editor plug-in of FUremote. Fig. 18 shows the streams of RDF data packets and the form in which they are shared and stored by the corresponding software packages.

4.1.1.2 Communication Establishment

The simulation package was not set up as an internal module of the FUmanoid control software, but rather as a form of central service provider, offering sensor emulation to the requesting remote programs using IPC.

Regardless as to whether it is used locally or as a server handling several remote requests, the connection process is equivalent. The connection establishment packets are also implemented as *Protocol Buffer* data structures, making it very easy to extend the interface later on, such as when requirements change during the development cycle.

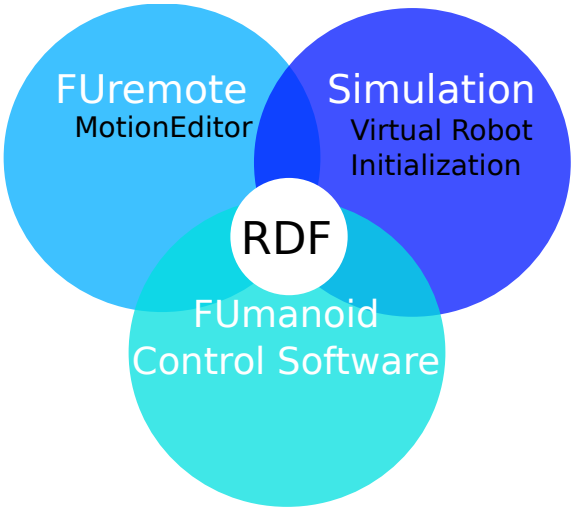


Figure 18: Robot Description Files are used in different FUmanoid software projects such as FUremote, control software and simulator

In Figure 19 the connection handshake between control software and simulation software is shown. First, the control software sends an initial request to register itself as a new client with the simulation software, which acts as a server. This request message includes information about the type of sensors needed for emulation and a reference to a RDF, which is used directly when creating the virtual robot. After the request is sent, it is answered by the simulation with a message consisting of sensor properties, such as port numbers. The control software then configures its adapters according to this information, e.g. the camera adapter of the control software will establish an IPC socket connection with the simulator.

One important requirement of the simulation software was flexibility in creating use cases. In order to achieve this, sensor emulation had to be completely optional. This means that sensors may or may not be attached to virtual hardware. Any combination of sensors has to be possible.

The *ClientManager* module receives requests from robot control software instances and initializes the process of creating a new simulated robot. Finally, this module hands the request to the *RobotBuilder*. This module is also responsible for reconnecting sensor ports where the connection was previously closed. The disconnection may be due to an error in the control software or may have been done on purpose when a patch had to be integrated. The control software therefore has to be stopped, rebuilt and started again. It is possible to pause simulation stepping during such a procedure.

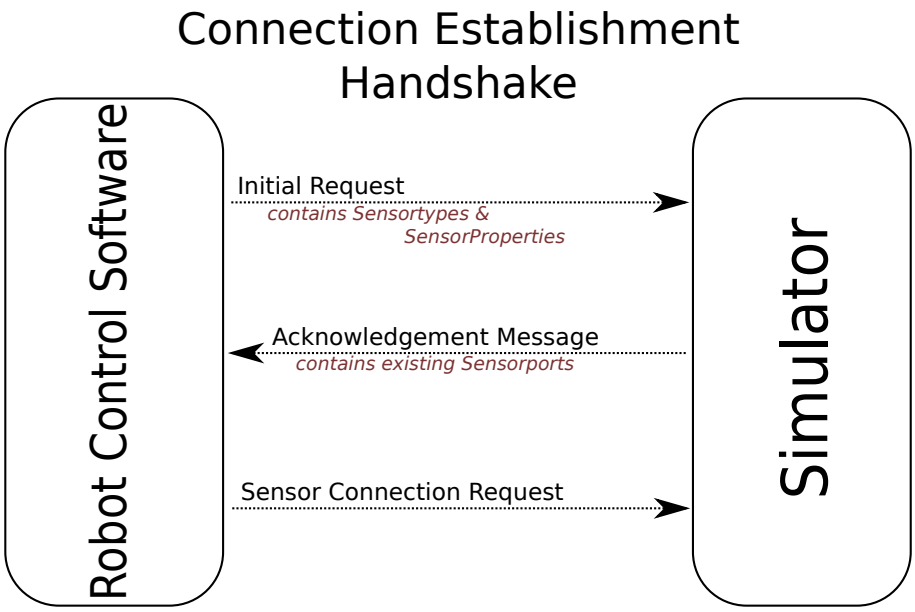


Figure 19: Time chart of a connection handshake

In order to create a broadcast-like behavior, even for locally executed test cases, a central hub is needed to duplicate and delegate *WorldModelStatus* packets from each robot instance. A sensor exchanging WorldModel messages can register itself with a *WorldModelHub* module, which then delegates all messages sent by its registered sensors.

The *RobotBuilder* module was built as a Factory Method pattern, generating objects of *Robot* derived classes. Each instance is created based on its RDF. A detailed description of the *Robot* data structure is given in Chapter 4.1.4

4.1.1.3 Sensor Message Exchange

Generated sensor feedback is sent to adapters, which transform the incoming data into structures known by the control software. The simulator has the generic *SensorData* type for storing this data internally. This is later converted into a *Protocol Buffer* packet in order to transmit it to the receiving adapter.

During the previously described connection establishment, all sensor properties, such as a specific lens distortion or a certain camera image type, can be directly set by the control software. In this way, hardware changes can be evaluated beforehand in a test case supported by generated sensor data. Furthermore, the same algorithm can be evaluated with different levels of noise to identify its level of noise tolerance.

For this configuration process, two different types of Protocol Buffer messages are exchanged:

Sensor Property Setting specific properties for a sensor is not mandatory. Each sensor type has its own default settings. This packet is sent once during the connection establishment handshake between simulation and control software.

Sensor Data A frequently exchanged data packet between an emulated sensor instance and its corresponding adapter within the control software. Alongside generated sensor feedback, this packet may also contain meta-data for debugging or learning methods.

S*ensorData* is also internally used as model data by visualizing elements of the *Graphical User Interface (GUI)*.

Interprocess Communication

In order to operate with any given and already existing interface it was necessary to support more than one method of *Interprocess Communication (IPC)*. This also offers additional options regarding how the simulation software itself is used. It integrates well when used in parallel with other FUmanoid software packages on the same machine, and furthermore can be used as a central simulation server within a *Local Area Network* (LAN). This allows it to be accessed by different machines or even the real robotic hardware running the control software with emulated feedback.

Three different types of sockets are currently supported:

- Unix Socket
- UDP Socket
- TCP Socket

Depending on the emulated sensing device, a continuous connection may be required (e.g. data streams) and in addition the set-up of the test session may also require different socket connections. The simulator can manage three communication scenarios.

Local Execution Simulation software and control software are executed locally on the same machine. The simulator has to manage the broadcast problems described before, but it is possible to use Unix sockets as IPC methods; this allows for high performance, especially when a large volume of data has to be exchanged.

Server The simulator acts as a server, and other machines running control software instances connect to it. In this set-up, Unix sockets cannot be used at all. They have to be replaced by TCP connections, which do not have the same performance due to the protocol overhead and transmission time.

Hardware in a Loop Comparable to the server set up, but this time the remote devices are the actual embedded systems, normally carried by the robotic hardware. In this case the simulator supplies the actual robotic hardware with generated data. The performance of an algorithm and its effects on the whole system can be evaluated.

4.1.1.4 Adapting to a Robot Control Software

The FUmanoid control software is evolving very quickly, to such an extent that the data representation was changed during the development cycle of this thesis. Consequently, the incoming sensor data has to be adapted by the control software and directly converted into an internally known format. This is done by implementing a virtual device such as a network camera device.

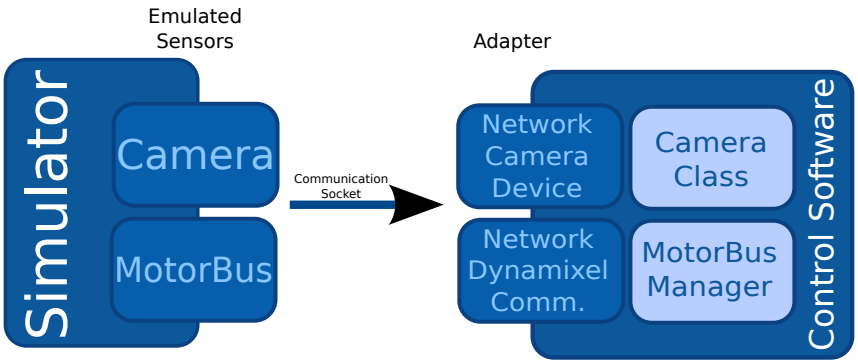


Figure 20: Adapter classes in control software

Currently three different adapter types have been implemented:

Camera This module can provide two different types of image. Firstly, YUV422 which was used in 2009. Secondly, image data stored in a Bayer Pattern. Their generation will be explained in Chapter 4.2.1.1.

MotorBus The FUmanoid MotorBus module allows different methods of data transportation, such as a serial connection or USB. In order to receive actuator data from the simulation, a network socket connection was added to the transportation layer. In this case no conversion is necessary, as only the original Dynamixel protocol is exchanged.

WorldModelStatus A message callback for received WorldModel status packages is registered in the communication module of the control software. Whenever a new packet arrives it will be dispatched to all listeners.

After being processed by one of the adapters mentioned above, the data can be treated by the control software just like sensor data from a real hardware device. Their use is transparent within the whole software package.

As a result of their generic format, sensor messages can easily be adapted by any other control software.

4.1.2 Dynamic Library Integration

Being committed to a dynamic package has several disadvantages. First of all, at the beginning of a development cycle, the requirement of a dynamic package may not be entirely clear. This has to be determined through the use of a precise validation process. Secondly, only having access to one single package makes it impossible to compare test results using a set of different libraries. Thirdly, while many dynamic packages are open source and freely available, their support is not guaranteed by anyone. Over the past few years many projects have been discontinued and others have been newly created. A generic interface to the physics implementation decreases the dependency on such uninfluenceable events.

It is also important to note, that in RoboCup ODE was predominately used in other implementations. Different dynamic packages have usually been evaluated in terms of their feature lists, but rarely due to poorer performance in a direct comparison with ODE in a RoboCup simulation scenario. This may have been due to the effort needed to study the different APIs, whereas ODE was known to work sufficiently.

As a result, an open source project called *Physical Abstraction Layer* (PAL) is used as an intermediary between the physical representation of a dynamic package and the internal representation of the simulation. PAL was developed by PhD student Adrian Boeing as part of his doctoral thesis. In 2007 he and Thomas Bräunl from the School of Electrical, Electronic and Computer Engineering of the University of Western Australia published an evaluation of real-time physics simulation systems. [10] PAL can be used on multiple platforms and supports Windows, Linux and MacOS. It is still well-supported by its developer community, even if some formerly supported engines have been dropped over the years. Currently, a wide range of different dynamic packages are supported [9]:

- Box2D Physics Engine
- Bullet Physics Library
- Havok^{TM} Physics - experimental support
- Impulse Based Dynamic Simulation (IBDS) - experimental support
- JigLib - Rigid Body Physics Engine $*$
- Newton Game Dynamics
- Open Dynamics Engine (ODE)
- PhysX (Version ≤ 2.8)
- Tokamak - Open Source Physics Engine $*$
- TrueAxis $*$

(*) - no further development for the marked libraries

Each physics library mentioned above has its own integrated algorithms. PAL offers a more abstract interface to access each underlying library in the same way. Due to its architecture, it is easy to add new dynamic packages or solve broken interfaces after a release of a new version. The interfaces of the established dynamic packages seem to be in good conditions.

The most common geometries, such as boxes, capsules, spheres, planes and convex meshes are supported, as well as three basic joint categories: ball, hinge and slider joints. PAL also provides a generic 6DoF link, but it is not supported by every underlying dynamic package.

On a higher level, sensor and actuator support is provided as well, but not every library is supported in the same way. Currently, the PAL implementation of the Bullet Physics Library seems to cover most of its features. In the case of a simulator for the RoboCup Humanoid League, only a small selection of PAL provided sensors were used: Contact Sensor, Compass, Gyroscope and a DC Motor.

An additional feature provided by PAL is its built-in benchmark suite to compare the performance of different dynamic packages. The contemporary results were presented by the authors in 2007. In his conclusion Boeing states that the Bullet Physics Library provided the best results overall with open source engines, outperforming even some of the commercial engines. Especially in the collision penetration test the performance of Bullet was significantly better than the other tested libraries [10].

There are however some drawbacks to the project. It is not possible to switch the selected physics engine at run time, and also there is no support for multiple engines within one scene. The first issue can be solved within the simulation software using

the abstraction layer itself by creating a completely new world based on a different physic library. The second issue cannot be solved, since it is not achievable by design.

Provided Libraries in the Simulation Software

With ODE 11.1, PhysX 2.8 and Bullet Physics Library 2.79, three dynamic packages were chosen to be integrated using the abstraction layer and all were briefly described in Chapter 2.4.1. Currently, only two of them are supported because PhysX does not work on a 64bit Linux system, which became a requirement because it became common in the FUmanoid development team. The PhysX SDK 3.0 provides 64bit support, but PAL is not supporting this version in its latest release.

4.1.3 A Simulation Case

The central data structure within this simulator is called SimCase. It can be seen as the root node: every other object will be created, initialized and destroyed within the module, and it includes the main simulation loop. Each object belonging to the scenery of a SimCase is created during the initialization phase, meaning each SimCase is a description of the physical world and the objects which make up its surroundings.

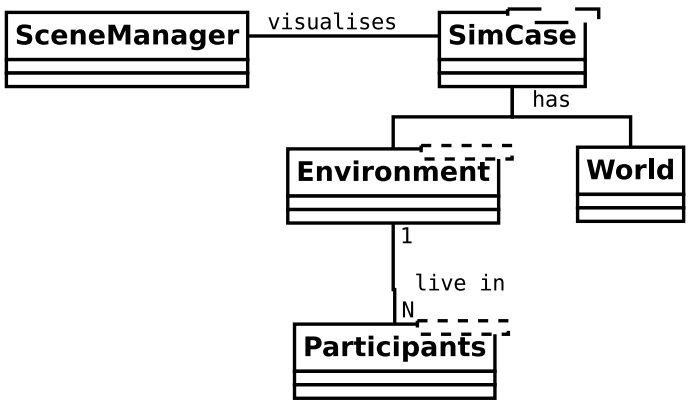


Figure 21: A class diagram showing the internal relations of a SimCase

4.1.3.1 World

As the SimCase is the main node for the whole simulation, the world module is the same for the dynamic stepping process. It is the central clock of the system, the

stepping process is started, paused and stopped here. Also the step length, which is the amount of time between two steps, is provided by *World*.

General properties can be chosen by the *SimCase*, which is the creating object of *World*. Currently, the main properties are:

- gravity
- artificial friction (for a more realistic ball movement in ODE)
- simulator clock settings

The world object is fully controllable by the SimCase object, which can deactivate physical feedback if necessary. For example, a GUI interaction may manipulate the current state of the environment. Therefore, the world's clock is stopped and the feedback is set to an inactive state until the user interaction has been completed.

Environment

World is a module, which is used with specific parameters in each *SimCase*. The required abstraction for defining objects that clearly belong to a single use case is provided by *Environment*. It can be seen as a container for all created participants and also as a factory for these objects (see section 4.1.4). The stored data is accessible for model-view-controller based GUI elements.

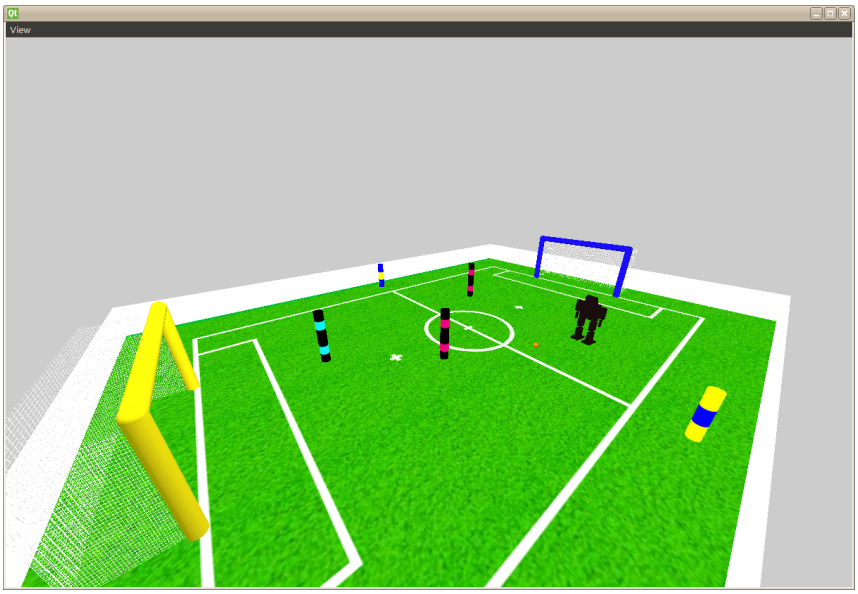


Figure 22: Soccer pitch with humanoid robot and obstacles.

For example, a FUmanoid *Robot* object is provided by an *Environment* describing

the RoboCup related surroundings in a *SimCase* handling all objects that are needed to simulate a Humanoid League KidSize match.

Environment is also responsible for a correct destruction of its registered participating objects.

4.1.4 Modeling of Simulated Objects

Simulated objects can be represented in several ways. Generally for this, compounds of geometries are used to form the physical representation of a *Participant* object. A *Participant* is a multi body system and the main interface of complex simulated objects towards the *SimCase*. It can be categorized into item groups, such as *Robot*, *Ball* or *Obstacle* in terms of the RoboCup context, which can be identified internally without knowing anything about their physical structure. At this level it is possible to apply external collision algorithms (see section 4.1.5), without using the underlying dynamic library. Moreover, this data structure allows direct access to the center of mass (CoM), position and orientation of the whole object. *Participants* can either be static or movable and consist of one or more *Body* objects (e.g. Box, Capsule, Sphere or Compound). The generation and management of dynamic *Participant* objects is briefly described in section 4.1.4.1. Static simulated objects exist to improve the visual impression of the scenery or to simply block the way as an obstacle, although they do not have flexible joints.

The smallest entity in the simulation is a *SimNode* object. *SimNode* is an abstract class, which must be derived by each primitive geometry class. The formerly mentioned *Participant* derives from the abstract class *SimTree*, which has a pointer to a root node as an attribute.

4.1.4.1 Tree-Structured Simulation Participants

Simulated objects can have different forms of interconnection, such as kinematic chain, star topology, partial loops or a tree structure. In this thesis *SimTree* objects represent the simulated object as a tree structure. This is comparable to the kinematic chain approach, but also allows branches at each node. The simulation forbids joints, which would connect divided branches of the tree. A multi-body system having N *Body* nodes has to have $N - 1$ joints.

Each *Participant* class is derived from a tree structured multi-body system base class. Figure 23 shows a general layout of such a *SimTree* object. The root node is

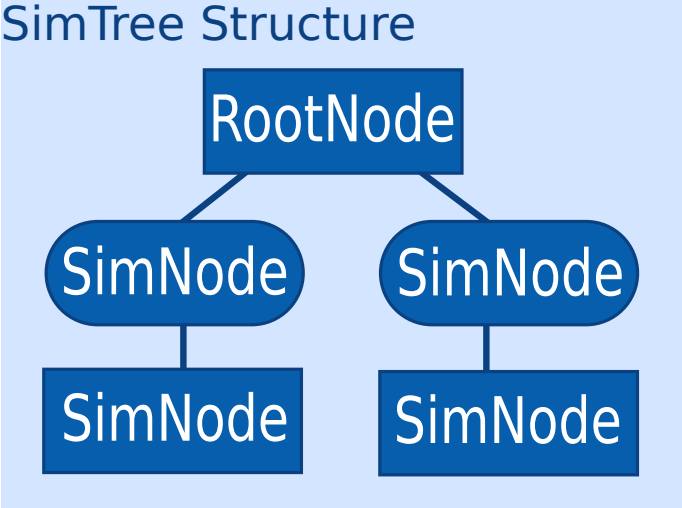


Figure 23: A tree-structured simulation object (A *Link* is visualized with rounded corners)

highlighted at the top of the diagram and the tree consists of two more types:

- BodyLink SimNode
- Body SimNode

For each type of node certain assumptions can be taken into account. Firstly, the leaves and root node of a tree have to be *Body* instances. *BodyLinks* are always inner nodes of the tree. Secondly, *BodyLinks* have to be embedded in two *Body* objects. A branch at a *BodyLink* node is not allowed, but several *BodyLink objects* may be connected to a single *SimNode*.

All properties of a *Body* object listed in Table 4 extend the information about the physical representation in the simulation process, which is managed by the underlying PAL object. The initial values have to be set during the model setup and in case of a robot may be globally defined in a RDF.

Attribute	Type
physicsEnabled	bool
world	World*
color	Vec3
material	Material
textures	Vector<int>

Table 4: Attributes of a basic *Body* object

The Body class has been derived from *SimNode* and the administrating tree ob-

ject can develop a high degree of complexity, such as a humanoid robot. The full representation of a FUmanoid robot is briefly described in the following section.

Robots As Tree Structures

A robot model is a compound of several *SimNode Body* objects connected by *SimNode BodyLinks*. The design shown in Figure 24 of a simulated autonomous FUmanoid robot has 21 degrees of freedom. The basic model has essentially stayed the same for each robot generation over the years, with the majority of modifications being to the hardware construction of the legs. This can be applied to the simulated model through the addition of further joint constraints.

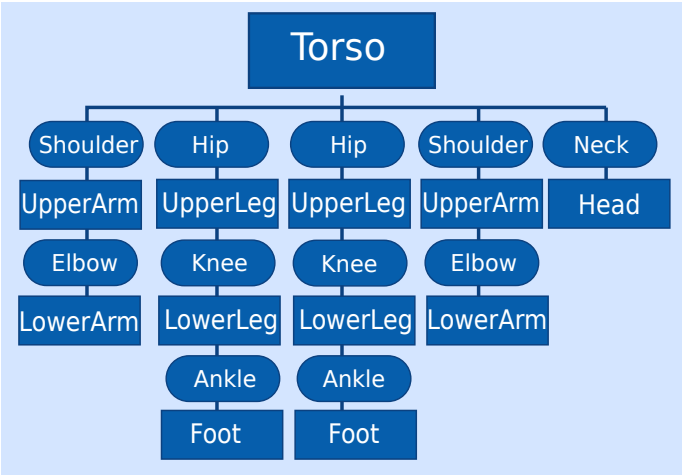


Figure 24: Example of a simplified variant of a tree-structured humanoid robot

In Fig. 24 all joints and their rotation axes are shown on the left. This data refers to the FUmanoid robot generation of 2009. Later on in the initialization phase, actuator sensors will be attached to all BodyLinks representing a certain type of servomotor. Their properties differ depending on the Dynamixel motor they are emulating.

4.1.4.2 Static Compound Objects

Static *Participant* objects are an exception. They are represented by a single *SimNode* containing a PAL *BodyCompound* object, which is a static bond of primitive geometries. A *BodyCompound* geometry still interacts with other physical objects, although no dynamic processes are evaluated within the construction.

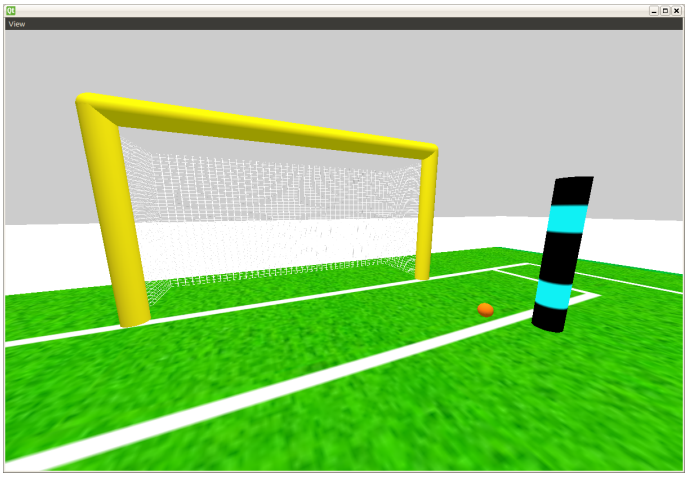


Figure 25: Visualization of a *Compound Object* (Goal)

Such elements are used for objects belonging to the robots environment. So they have to interact with other objects in case of a collision, but inner processes and forces can be ignored. For example, in the FUmanoid *SimCase* the surrounding advertising boards and the goals including posts and net are based on *BodyCompound* objects. For many dynamic libraries this reduces the computing effort and stabilizes the bonding of the compound itself.

4.1.5 High-Level Collision Detection

In an early stage of the development process the PAL version used was not able to detect collisions between dynamic and static ODE objects as well as only static objects. As a result, high-level collision detection was implemented for the virtual tree-structured robot models in order to pass the collision information to the engine by creating the physical forces artificially. Later this problem was solved by proposing a patch to the PAL development team.

Compared to built-in collision detection in ODE, the simulation software itself checks in every time step whether a tree-structure has any contact with other models or ball objects. Therefore, two different approaches have been implemented; they are similar in their behavior, but differ in their detection method.

4.1.5.1 Collision Detection Using Bounding Spheres

The main idea of this solution is very straight-forward. Each participating model is enclosed by spheres or a single sphere and every time step an algorithm checks

whether sectional planes between two spheres of different models exist.

Algorithm 4.1 Bounding Sphere Collision Detection

```
1: bool function bSphereTest (vec3d object1, vec3d object2)
2:
3: vec3d relPos := object1.pos − object2.pos
4:  float distance := relPos.x^2 + relPos.y^2 + relPos.z^2
5:  float minDist := object1.radius + object2.radius
6:
7: return distance ≤ minDist^2
8:
9: end function
```

4.1.5.2 Collision Detection Using Bounding Boxes

Similar to the first approach this one basically does the same, but using bounding boxes instead of a sphere. The advantage is that a box covers a humanoid robot much better than a sphere. The sphere results in many hollows since the enclosed body parts cannot be covered exactly.

Since this problem cannot be solved with one simple comparison, the *separating axis theorem* is used. Given two polyhedrons A and B. A line for which A and B have disjointed projections is called a *separating axis.* In the case of two rectangular solids this test has to be applied 15 times, because each has three linearly independent flat surfaces and three linearly independent edge directions, which leads to (3+3)+3∗3 = 15 tests. If there is only one test, which shows that there is no overlapping, then the rectangular solids are disjointed [32].

4.1.6 Data Visualization

Thus far, the generated data streams have been transported to the control software for further processing and analysis. In order to provide good data quality the user of the simulation has to be able to trace the current output. In this thesis the *Qt* user interface framework is used for graphical visualization; it is platform-independent and freely available [31].

Every *Participant* is registered in a global menu within the GUI, and each of its sensors is selectable. This *SensorData* is then presented in a Qt view separately in its own window. These widgets can be used not only for observing, but also

for manipulating the selected sensor data. For example, the goal position for the controller of a servo motor can be changed, thus affecting the pose of the robot.

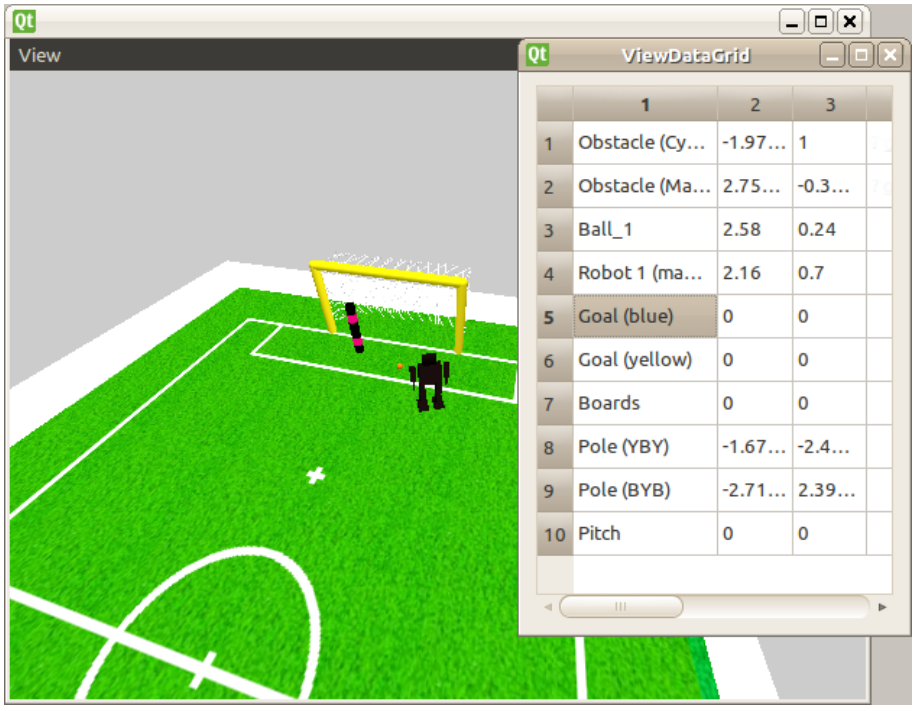


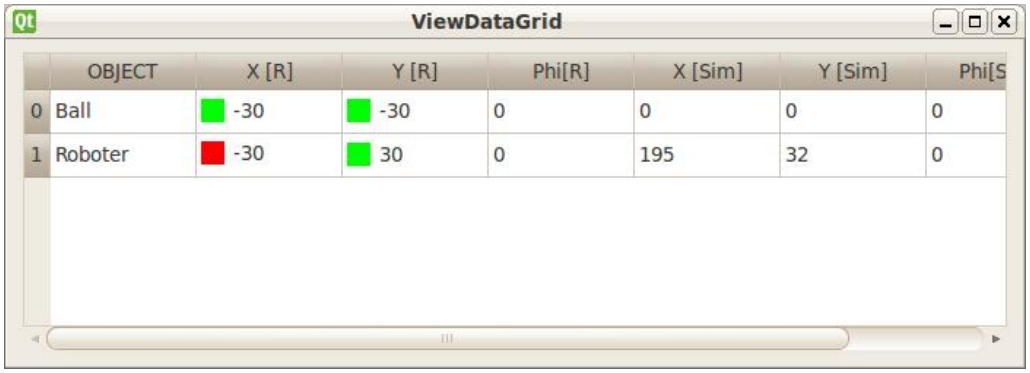
Figure 26: Overview of environment data on a grid in a Qt widget

Qt comes with very detailed documentation and is well supported by an active developer community. The design of Qt classes used to visualize data sets is based on the model-view-controller concept [31]. This strict separation of the data itself and its presentation is needed in order to guarantee that the visualizing classes are unable to manipulate the data set and instead call methods in the controller to do this for them. This results in a GUI that only consists of code necessary for display tasks.

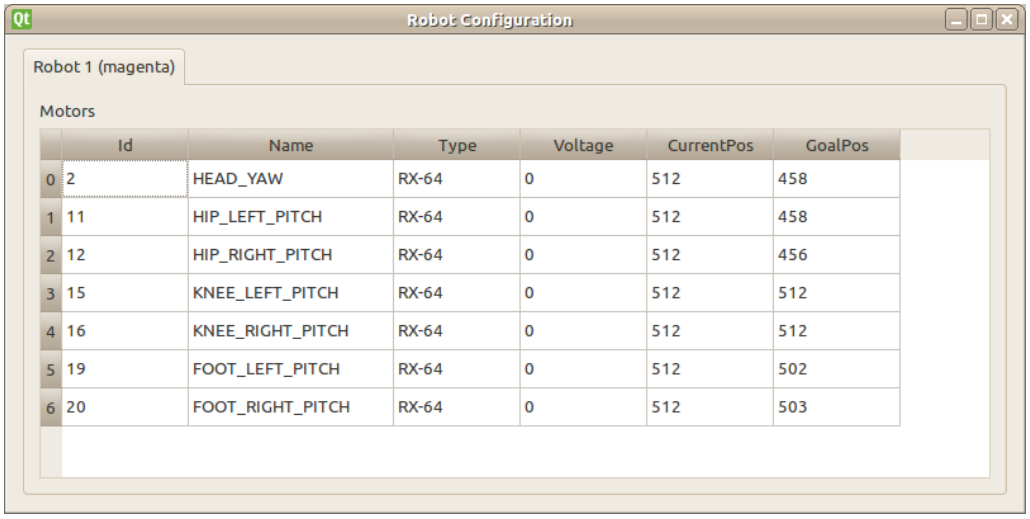
4.2 Sensor Emulation

This chapter introduces all virtual devices, which have been developed for the simulation software, as well as distinguishing between sensor devices which have an equivalent in the real world and virtual sensors. Virtual sensors do not provide typical analogue feedback, but data on a higher level of abstraction.

Section 4.2.1 describes the general implementation of an interface for emulated camera devices, presents two types of virtual Dynamixel actuators and introduces an



(a) The current localization quality is presented in a Qt widget



(b) Servomotor properties are presented in a detailed grid view.

Figure 27: Two different sensor status views.

IMU implementation which provides pitch and roll data related to its attachment on the humanoid robot model. It also details the foot pressure sensors which have been developed and the only virtual sensor which provides specific information about the game state.

4.2.1 Sensors for Humanoid Robots

In order to fully replace a humanoid robot it is necessary to replicate the behavior of its sensing hardware devices. Additionally, compliance is required with the definitions given by the unique communication protocol of each sensor in the case of very low level sensor abstraction.

Distinctions are made between two types of emulated sensors:

Local Sensors When the generated feedback is not directly sent to a certain destination, the sensor can be seen as acting locally. This is used to emulate devices which are hidden by a serial bus connection. The presence of such devices is not detected by the robot control software, as no explicit wired connection between them exists. In this case the serial bus connection itself is seen as a remote sensor and all devices sharing this bus connection are local sensors.

Remote Sensors A remote connection is essential for exchanging data between the emulating sensing device and the data receiving robot control software. In this thesis remote sensors use IPC to exchange data in the form of a Google Protocol Buffer data structure.

Furthermore, two special forms of remote and local sensors do exist:

Sensor Sets As mentioned in the description of the local sensor many local sensors can be hidden behind an interface derived from the *RemoteSensor* abstract class. *SensorSet* is exactly this kind of container and can ideally be used to behave like an interface with a larger set of sensors.

Sensor Set Item A single item which is registered at a *SensorSet* is known as a *SensorSetItem*. This abstract class provides the *receive()* and *respond()* methods to convey messages from local sensors to the *SensorSet* container, which then can pass the messages on like a normal remote sensor using IPC.

As a result it is possible to design a sensor on any level of abstraction. The output data of a camera sensor can vary from raw image data to the results of pre-processed images, such as correctly detected and labeled objects. New layers of data abstraction can be added to the specific sensor data structure.

Sensor devices can easily be added to any tree-structured robot model. For example, if it is not necessary to simulate the robot's body movement and only project its behavior on a soccer pitch, no actuators have to be attached to the formation. This also increases the simulation performance by reducing the need for processing resources. Furthermore, this layout is completely independent of the chosen type - reactive or deliberative - of robot control software.

Easy integration of newly-developed sensors was one of the key requirements. Sensors do not share any important dependency, so at any time sensors may be removed or new virtual sensor devices may be added to the simulation software.

4.2.1.1 Camera

This section describes the representation of a generic camera device in the simulation software. The implementation of the initialization, configuration and update process in each time step of such a device was realised in this thesis, this was necessary in order to emulate many different types of cameras and camera lenses. This section excludes all aspects of the camera lens model implementations and additional noise and distortion filters for emulated camera images, which are covered in Sebastian Mielke's diploma thesis.

The *Camera* abstract class derives from the *SensorRemote* super class and can be seen as an interface for all sensing devices that are required to deliver a series of images in a configurable frame rate. By default 20 images per second are "captured" and transferred to the remote destination. Different lens models can be attached, but have to be set up during the configuration process. As of yet, a change of lens models after initializing the camera is not supported. The color code can also be chosen, with the simulator historically supporting the RGB and Bayer Pattern color codes as well as YUV422 for current models.

The camera is "mounted" on a *Body* object, which establishes direct physical feedback for the camera device, thus allowing it to automatically move correspondingly when the head is turned by an actuator. This is done by using a Viewpoint object, as described in the following section.

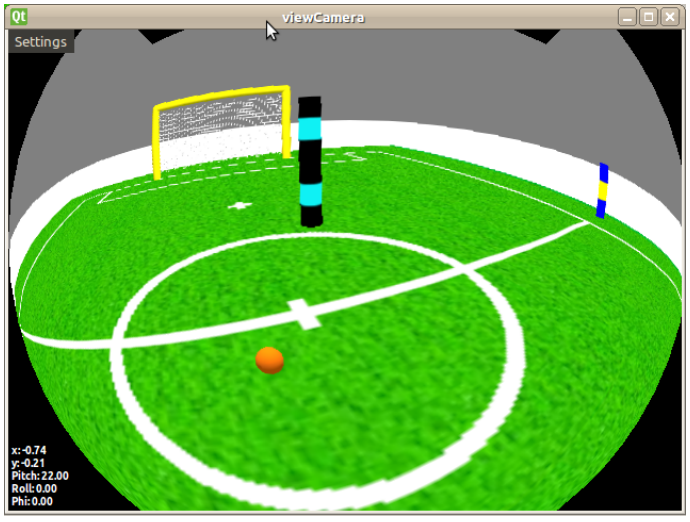
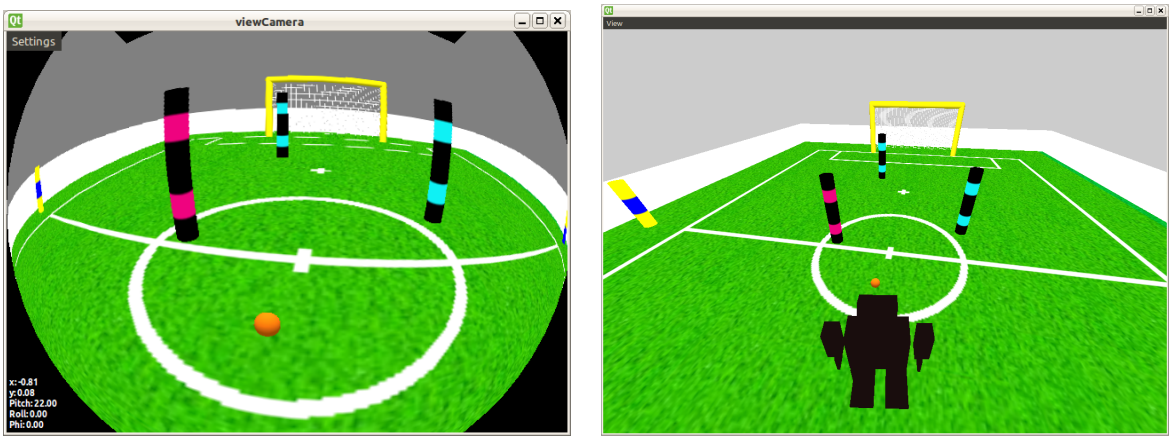


Figure 28: Example of a generated camera image with a fish-eye lens

Viewpoints

As a representation for camera positions within the physical world a ViewPoint instance can be attached to any physical body object or move around freely. The two possible types of viewpoints are as follows:



(a) (b)

Figure 29: The images show two types of *Viewpoints*. The linked type in (a) and the movable type in (b)

Movable Viewpoints A freely movable *Viewpoint* is not related to any other objects and can be moved by keyboard and mouse control in a Qt widget providing the user interaction events for this method. Since this kind of *Viewpoint* is not connected with the physical world, no feedback has any impact on the camera position.

Linked Viewpoints This type of *Viewpoint* is directly linked to a *Body* object within a *SimTree*. It cannot be manipulated by any input device. Each manipulation is related to a physical force moving and rotating the Body object.

Viewpoints are the connection between the physical representation layer and the visualization layer. It is the only interface where physical forces have an impact on the resulting visualization output this is the case with a *LinkedViewpoint*.

The camera class representing a special type of camera and lens, will always be implemented as a linked version of a Viewpoint. The generalized 3D overview representing the overhead camera in the simulator is a freely movable Viewpoint.

4.2.1.2 Actuators

A humanoid robot is defined by its ability to walk bipedally. As such, a central part of the development process is a strictly controlled interaction of actuator movements provided by a dynamic walking algorithm. Besides the robot's design itself and its mass distribution, the actuator properties have a huge impact on the control mechanism.

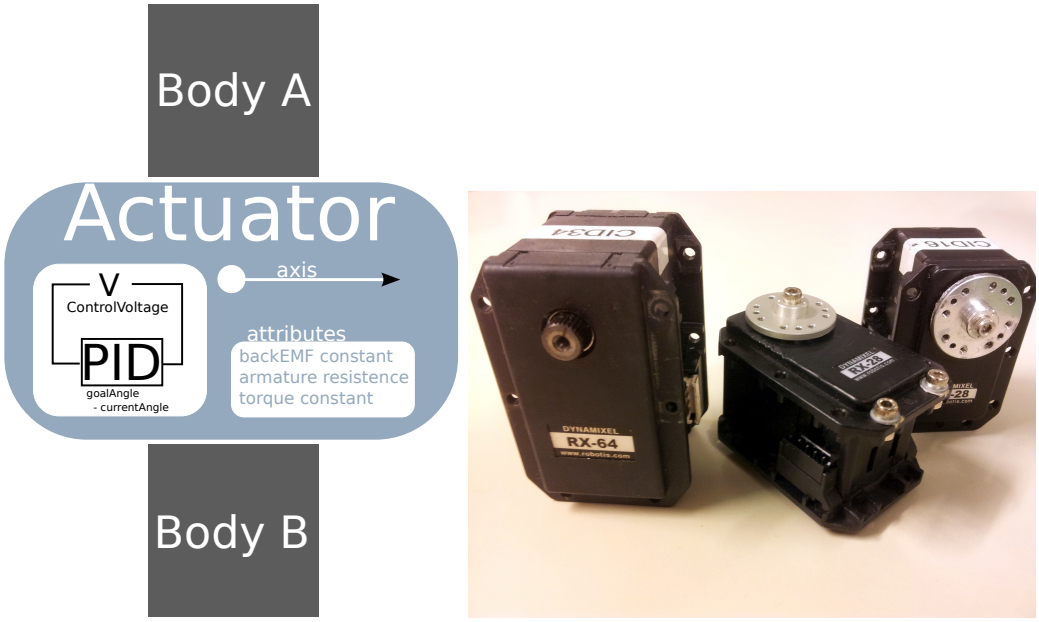


Figure 30: Implementation of an actuator: Concept and real hardware device

In Fig. 30 a type RX-64 servomotor is shown. The FUmanoid robots currently consist of 14 actuators of this type. Additionally, seven servomotors are of a smaller and lighter, but less powerful model - called RX-28. The specifications of both models are listed in Table 5. The difference between the two types is that the RX-28 is able to turn faster, but provides less holding torque. Furthermore, it is smaller and lighter than the RX-64, which makes it easier to integrate into a small humanoid robot construction [34, 35]. The FUmanoids use both types with the same operating voltage, which is always between $14.8V$ and $16V$. Consequently, the RX-64 model always operates under-capacity, which becomes apparent in the motor's reduced velocity.

The class *DynamixelServo* is designed as a *SensorSetItem* - a sub-type of *SensorLocal*. In order to create a comparable behavior of the virtual servomotor with its role model in reality, it is necessary to calculate a few of the characteristics of both

	Dynamixel RX-28	Dynamixel RX-64
Weight	72 g	125 g
Size	36 x 51 x 36 mm	40 x 61 x 41 mm
Operating Voltage	12-16 V	12-21 V
Holding Torque	37 kgf cmA	52 kgf cmA
Velocity	67 RPM	64 RPM
Connection	RS-485 Multi Drop Bus	RS-485 Multi Drop Bus
Comm. Speed	7343bps ~ 1 Mbps	7343bps ~ 1 Mbps
Sensor (motor position)	Potentiometer	Potentiometer
Resolution	0.29°	0.29°

Table 5: Properties of Dynamixel RX-28 and RX-64 actuators [35, 34]

types.

Throughout the whole process of emulating the correct behavior of a Dynamixel servomotor *DynamixelServo* is used for motor register management and additionally for dynamic position control. The robot control software will send commands in order to set a joint to a certain position. A DC-controlled servomotor needs to be supplied with a certain amount of voltage in order to hold or reach a position. The further this position is from the starting position, the more voltage is required. This is done internally by a controller within the motor device. In the case of the simulator, a PID controller was implemented to emulate the Dynamixel behavior as closely as possible.

Controller output is defined as

$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{d}{dt} e(t)$

where K is the specific PID tuning parameter (K_P: proportional gain, K_I: integral gain, K_D: derivative gain) and e is an error function giving the error value, which is the difference of the chosen set-point and the current process value, at a certain timestep t. The tuning parameters have to be set and adjusted properly, since controlling mechanism tend to overshoot and this may result in oscillating states. Therefore, a heuristic method, formally known as the *Ziegler-Nichols Method* was chosen*[43]*. The values of K_p,K_I and K_D are no longer set individually, but are related to an ultimate gain parameter K_U and an oscillation period P_U as shown in Table 6. Initially K_U is determined by setting K_I and K_D to zero and only increasing the P gain until it reaches the ultimate gain K_U, at which at the outputs start to oscillate.

The *DynamixelServo* class manages a *BodyLink* instance with an active *DCMotor*.

Type	K_P	K_I	K_D
PID	$0.6K_U$	$2\frac{K_P}{P_U}$	$\frac{K_P P_U}{8}$

Table 6: PID parameter tuning based on the Ziegler-Nichols method [43]

The emulated servomotor is based on a general model for dc motors. Although, the Dynamixel servomotors consists of additional components such as gears, the setup can still be treated as a black box behaving like a dc motor model. This assumption is true, when the dynamics of gears is shifted to the dynamics of the servomotor's rotor. The used model does not take weight, rotational inertia of gears and the inertia of the rotor into account. It has three important characteristics for its definition:

BackEMF (Vs/rad) constant Also known as the counter-electromotive force the backEMF is the force, that pushes against the current which induces it. It is caused by a changing electromagnetic field. The backEMF constant k_e of a motor is specified in the data sheet of the motor's manufacturer. $V_{emf} = \omega k_e$, where ω is the angular velocity of the motor when torque is zero and $V_+ = V_{emf}$.

Armature Resistance (ohms) The initial current will be limited only by the resistance of the armature winding and the brushes. At that instant the backEMF is zero and the only factor limiting the armature current is the armature resistance. Most likely the armature resistance of a motor will be less than 1Ω. $I_S = \frac{V_+}{R}$, where I_S is the stand-by current.

Torque Constant (Nm/A) The torque constant k_t of a motor describes the coherence between the torque τ and the current I. Where $\tau = I_S k_t = V_+ \left(\frac{k_t}{R}\right)$, which means that the current is proportional to the torque.

All of these characteristics can be derived from given data of the Dynamixel data sheet and can then be calculated using the equations above:

- From RX-64 data sheet: $V_{emf} = 15V$, $\frac{1}{\omega} = 0.188\frac{s}{rad} \Rightarrow \omega = 5.32\frac{rad}{s}$, $I_S = 2A$ and max. torque $\tau_s = 64kgf.cm$
- This gives us the following values using the equations above: $k_e = 2.82\frac{Vs}{rad}$, $R = 0.75\Omega$ and $k_t = 3.2\frac{Nm}{A}$

The DCMotor instances are initialized with the values from Table 7 in order to obtain a similar servomotor characteristic.

Motor Type	BackEMF	Armature Resistance	Torque constant
Dynamixel RX-64	$2.82\frac{Vs}{rad}$	0.75Ω	$3.2\frac{Nm}{A}$
Dynamixel RX-28	$2.02\frac{Vs}{rad}$	0.8Ω	$1.9\frac{Nm}{A}$

Table 7: Properties for virtual Dynamixel actuators

Representing the MotorBus Structure

The MotorBus is a structure within the FUmanoid control software that manages all communication with linked devices on the serial bus connection using the Dynamixel protocol. There are two types of messages in this protocol [35]:

- Instruction Packet
- Status Packet

A typical instruction packet is used to request a device to follow a certain instruction. The structure of this packet is shown in Table 8.

Head	ID	Length	Instruction	Param1	...	ParamN	Checksum
0xFFFF	1 Byte	1 Byte	1 Byte	1 Byte	...	1 Byte	1Byte

Table 8: Structure of a Dynamixel instruction packet [35]

Each message begins with two bytes containing the hexadecimal values *FF FF*, followed by the ID of the requested or answering device and the length of the complete packet, which is calculated as the number of parameters $(N) + 2$. The messages are never split into two parts.

The status packet is very similar, but is only used by devices to answer an instruction packet, e.g. sending the register values after a READ instruction packet. Besides the instruction field, the status packet contains a byte for an error code. If errors occur, they are decoded in the fifth byte, such as motor overload, overheating or undefined instruction errors.

Head	ID	Length	Error	Param1	...	ParamN	Checksum
0xFFFF	1 Byte	1 Byte	1 Byte	1 Byte	...	1 Byte	1Byte

Table 9: Structure of a Dynamixel status packet [35]

The check sum field is filled with the result for the follow equation:

$csum = \sim (ID + Length + Instruction + Param1 + \ldots + ParamN)$

The emulated actuator device only reacts to request commands and will never send packets without being requested to do so beforehand. There is no continuous still-alive or ping message that signals the devices existence on the bus.

The virtual Dynamixel devices have to understand and react to the most important commands, which are used to control all devices on the MotorBus. Three of them are more closely described as follows [35]:

READ This command is used to read certain registers. After the command is processed by the addressed devices, a status packet (see Table 9) containing the requested register values is generated.

WRITE This packet is used to set selected registers. In the case of a servomotor, a new goal position could be set in order to move the motor. No acknowledgment packet is sent after having received and processed this message.

RESET All settings are deleted, including the device ID and its current register values.

The MotorBus structure has strict timeout preferences and expects to have very low latencies. Compared to the serial connection in reality the virtual devices use a UnixSocket connection for communications. Currently, with many devices connected to the bus topology there will be timeouts for a couple messages. For the physical hardware, a return delay time of $0.5ms$ is the default value.

4.2.1.3 Inertial Measurement Unit

In 2010, the FUmanoids used a 5 DoF IMU which incorporates a dual-axis gyroscope and a triple-axis accelerometer on a small board. The IMU device of the FUmanoids is integrated in the robot's hardware system as a part of the serial bus, which is normally used for servomotors only. It was the first bus extension that was not an actuator. The IMU provides pitch and roll angles of the robot's body pose in 50Hz intervals by using a Kalman filter after combining selected outputs from the gyroscope and accelerometer.

Its virtual counterpart is set up to behave in the same manner and can be attached to any *Body* object using its axis and rotation matrix to calculate its current position.

A problem occurred when using the virtual IMU in the simulation, if the virtual device was positioned differently to its position in reality. Therefore, an offset needed to be distinguished and stored as a device property. Furthermore, the zero position of an IMU is not necessarily one of the robot's idle positions; it is possible to set

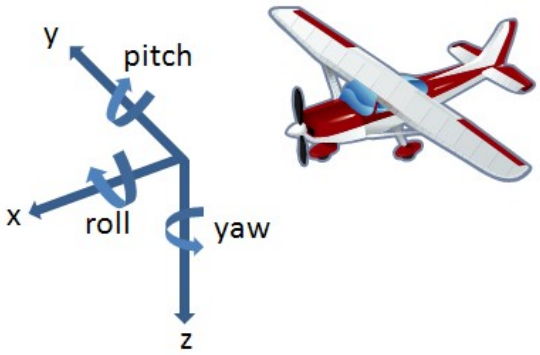


Figure 31: The three axes of an inertial measurement unit (pitch, roll, yaw)[g]

an offset in the simulation to compensate for these inaccuracies. In the first model of 2009 the head was rotated about the Y-axis through approximately 22 degrees. In the simulation only one body object existed and so the IMU had to be attached to the pre-rotated body object, meaning that normally the pitch-value would have been -22 degrees. However, the vision system corrected this rotation on the fly and assumed the IMUs pitch value to be zero degrees, since it was in reality attached to the neck and not to the head of the robot. Therefore, an offset was used to reset the value to zero in the simulation as well.

4.2.1.4 Foot Pressure Sensors

The binary ground contact sensors were added in 2009 as an additional extension to the serial motor bus. Later in 2011, this device was replaced by a board unifying four load cells and providing their current state of pressure using the known Dynamixel protocol.

The implementation of a binary ground contact switch was effected by attaching it to a corner of one foot's ground plate - a ground plate is one side of a *Body* structure represented in PAL by a palBox object. If any collision occurs between the ground terrain plate and this body object, it is straight forward to identify which of the four switches has made ground contact, as the position and rotation of the pal object is known.

This device also communicates using the Dynamixel protocol. It was therefore necessary to implement the foot switches as Dynamixel devices to preserve the requirement by the robot control software.

One drawback of this implementation is the frequency with which read commands poll the device for information. Even under lower load the simulation is not able

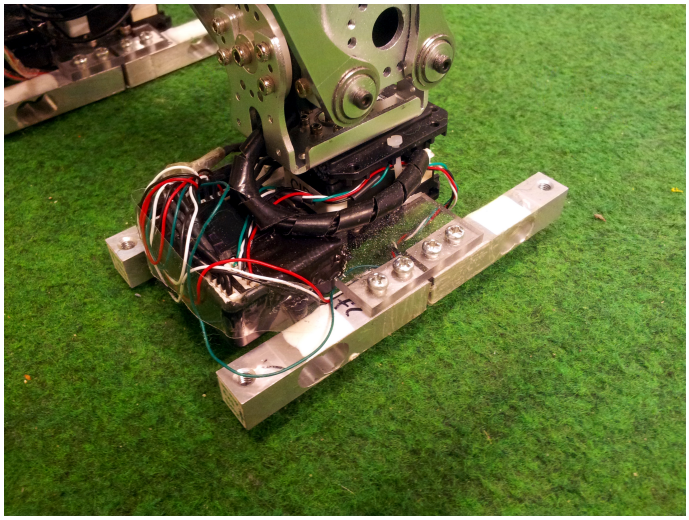


Figure 32: Image of a load cell device used as a foot pressure sensor

to provide more than $50Hz$ on the representation of the serial bus. The control software does not cache values, but asks directly for sensor outputs. As a result many read errors occur for this type of sensor, and this leads to abnormal execution of the current walking algorithms. Usually the walking algorithm is runs at over $75Hz$.

4.2.1.5 Virtual Sensors

All previously described sensors replace a device which does exist in reality. Virtual sensors have no such counterpart, and there is no restriction to the data they can provide. Furthermore, this group represents fictional sensing devices. As the simulation software is 'omniscient', it can support an attached device with ground truth data or combined data sets, which would not be obtained by a single device for the robot control software in reality.

Virtual sensors therefore allow a new form of simplification in a test scenario, which offers many new opportunities for multi-level testing. A virtual sensor can provide any data that is needed on a certain layer of data abstraction within the control software. No matter what kind of modules would usually pre-process the data, they can now be bypassed completely and will no longer interfere with the input data stream. This guarantees that all faults found in the output data were added by the tested modules themselves. Furthermore, it is easier to compare a simple pair of input and output data instead of tracing a failure within a long chain of several

inputs and outputs from different modules. In addition, this method can also be used in combination with distortion where a controlled amount of noise is added to the former ground truth data.

For example, in order to increase the quality of the FUmanoid behavior module, a virtual sensor was created to provide a full set of world model data to the control software, including correct vision and localization data sets. The bypassed modules (vision, localization, network data fusion) can then be excluded as potential sources of failure in the testing process.

Iterative Worldmodel Exchange

Rather than allowing the control software to extract all objects out of simulated images and pass this information into the localization module, things can be simplified greatly by replacing all the world model data with ground truth data from the simulation software. The result of this is that no errors from modules other than the behavior module can affect the behavior control. Figure 33 shows a detailed diagram of the described procedure.

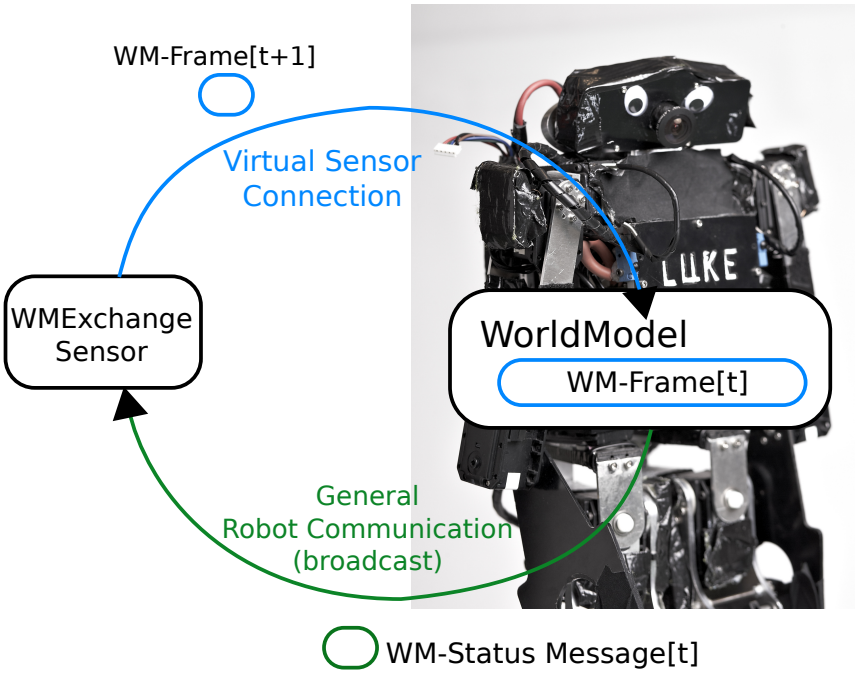


Figure 33: Diagram of an iterative world model exchange process

The placing of the whole world model entity within the control software instance requires many fields to be filled with meaningful data that correctly describe the

situation and the environment of the robot in its current position. For example, the following information is exchanged:

- global position of the robot itself
- global position of the ball
- global position of landmarks (side-poles, field line circle)
- global position of goal posts
- global position of obstacles

As mentioned before, all information is 100% accurate, since it is the world's ground truth data. As a result, the performance of the behavior module can be seen as totally unaffected by surroundings. When using the virtual sensor in this way, many tests, which are usually very difficult, become very straight forward. For example, observing tactical behavior of more than one robot or evaluating team work to solve complex situations.

By doing this the robot control software gains a complete set of ground truth data. In order to create a more realistic set of world model data, certain constraints have been applied to the virtual sensor:

- field of view is restricted to less than 180 degrees
- walking speed is limited to 40cm/s
- only the information about the nearest ball is sent to the control software

As well as constraints being very important, to create more realistic data sets, the addition of noise is also beneficial. By applying distortion filters the data quality can be reduced in a controlled manner. For example, this is a good method for evaluating data fusion processes such as the network information. A FUmanoid robot receives information about the game state and the state of the other robots via WiFi. Data can be extracted in order to evaluate how much noise can be tolerated while still being in a position to successfully perform team work tasks.

Since all data is provided by the physical world anyway, no additional performance overhead is necessary to generate the virtual sensor's feedback.

5 Experiments with Simulated Data

In this chapter all restrictions mentioned in Chapter 3 are reconsidered and the question as to whether the implemented simulation solution is in fact an enhancement of multi-level testing scenarios is discussed and evaluated through experiments. The simulation was therefore used with various approaches and differing levels of intensity. In the following sections a closer look is taken at the three biggest development tasks and how their development could be supported with the simulation package. All experiments were validated on a laptop with an with an Intel Core i3 CPU clocked at 2.13 GHz and 4GB RAM.

Additionally, this simulation software was used in two software project courses at Freie Universität Berlin. The students faced different fields of application, but rather than using robotic hardware, only the simulation software was provided to develop their software solutions.

5.1 Computer Vision and Self-Localization

In 2011, the emulated camera and its fish-eye lens were already used to evaluate changes to the vision system or sent ground truth data for the localization module development.

There are a variety of reasons why, at this stage, the simulation is unable to satisfy the need for an image database described in section 3.1.1. First of all, in the initial version the emulated images had pure colors and no noise at all. Thus, each camera image was perfectly color-calibrated and further processing steps came across no problems in extracting the correct objects. These kind of images were therefore useless for challenging the tested vision module. Secondly, the quality of field line features at greater distances created problems that did not exist in real camera

images. All these challenges are covered in detail in another diploma thesis as previously mentioned.

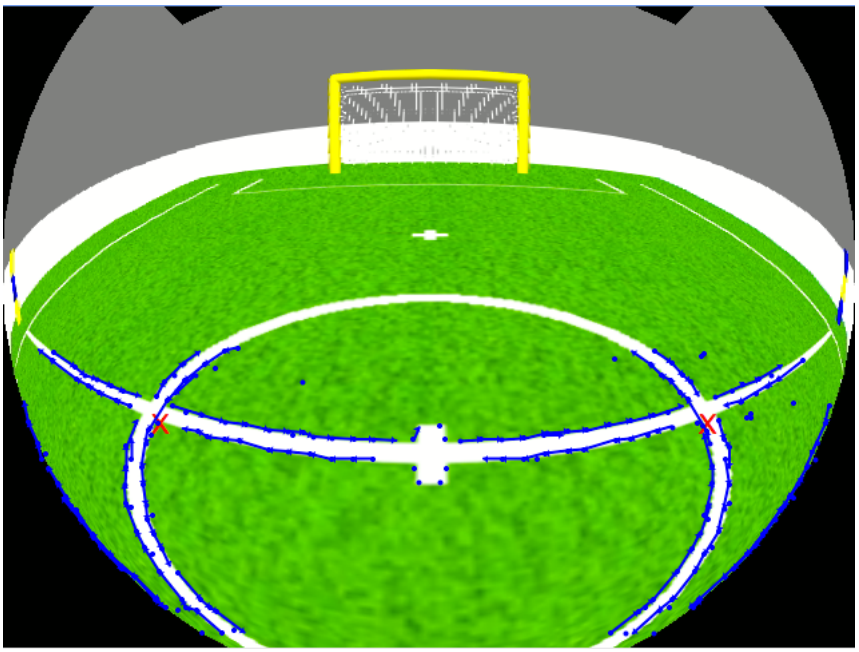


Figure 34: Generated image with overlays showing detected edges and field line features (e.g. a crossing (X))

In the aforementioned software project course in 2011, not only the behavior control was developed by students. Since the simulation software is able to provide all kinds of sensors at this point, students were able to choose any kind of field of application from within the KidSize Humanoid League. They included vision-related topics such as extracting a ball from raw image data without using colors, or trying new approaches to model the robot's environment. A result of a field line feature extracting algorithm using detected edges from a generated image is shown in Figure 34.

Although the images are not comparable to real images, the simulation was used by all students needing vision data. Due to the heavy distortion of a fish-eye lens, a different lens was used compared to last season's. When the students began their work the new lens was not attached to the new robot model for RoboCup 2012, which meant no image database existed for their testing purposes. The new hardware setup was reproduced in the virtual world within one working day and even though the images were not able to provide challenging vision tests, they fulfilled the initial requirements of the development cycle, when most students struggle with

the implementation. Later on in the process, when the hardware is ready, an image database is produced and the project participants are able to test their modules in a hybrid approach, swapping between emulated and real data.

5.2 Behavior Control

As part of a software project course, a group of students was tasked with developing new behavior strategies for the FUmanoid robotic agents. Principally, they were allowed to make modifications to the XABSL behavior code in order to create new roles or tactical sequences. The students also had access to the C++ code base to implement more complex functions such as probabilistic filters or other methods requiring more complex calculations unable to be solved in XABSL itself. At this stage the simulation software was still in the early stages of development, and as such could only provide basic ground truth data as ideal world model data.

The participants formed groups of two and developed their own behavior control in a time frame of four months. The only restrictions made by the tutors were based on the current version of the RoboCup rules, so tactical team play had to be done with three robots. In the end, five student teams managed to create a working version of their behavior and took part in a tournament at the end of the semester. In order to compare the results of the teams with the latest FUmanoid behavior it participated in the competition as well.

Interestingly the winning team also beat the latest FUmanoid behavior, which made clear that behavior control created only in the simulator includes less exceptions necessary to control a real humanoid robot. This advantage allowed the behavior developed in the student project to react faster and the game play looked more determined, but it is not possible to achieve the same results with a real robot without adding important exceptions such as overshoot protection to avoid instability to the developed code.

Currently, the simulation package is being used again to provide artificial sensor data in a programming software project as in semester one of 2011. The technical challenges are also prepared in the simulation software before testing in a real scenario with the new hardware setup.

Additionally, the whole behavior control concept of the FUmanoids was implemented for RoboCup 2011 by a single person using the simulation software. This was a genuine improvement compared to the previous years. In 2010, three people worked

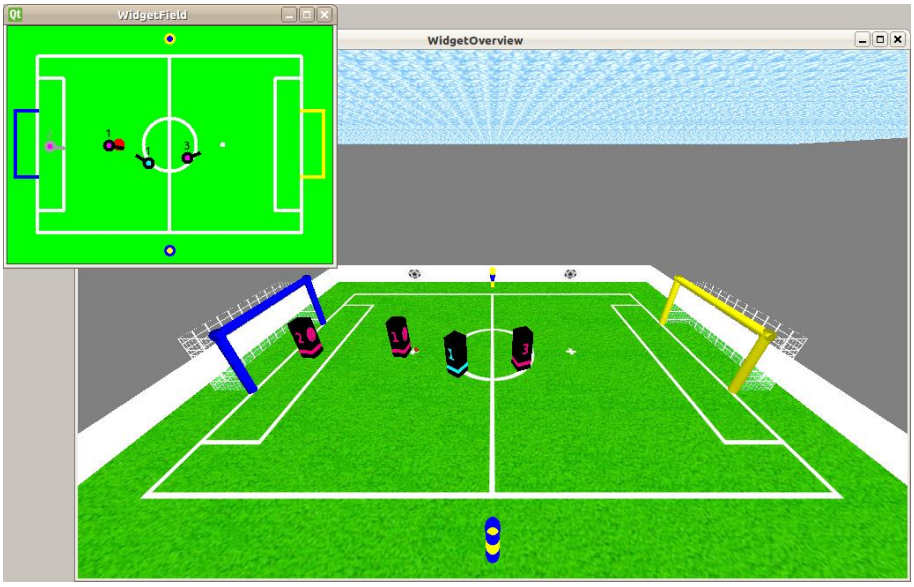


Figure 35: Early version of the simulation software used for a behavior control tournament in a software project course

on the behavior and strategy layer and in 2009 as many as five people were in charge of special robot behavior roles.

Especially in terms of the problems stated in section 3.1.4, using the simulation has seriously improved the work-flow for behavior development. Complex scenarios such as team play behavior tests can be set up within seconds and evaluated by a single person. Likewise, it is possible to control up to six robot agents using a mouse or keyboard, a test situation that is not possible in reality when robots in an early development stage tend to suffer mechanical malfunctions. In the case of a virtual humanoid soccer match, these situations can easily be solved by pausing the simulated worlds stepping mechanism and placing each robot in a safe position.

#Robots	System Load	Simulation Steps / Sec.
0	0%	98.2
1	<1%	96.7
2	4%	94.8
4	6%	90.5
6	10%	84.6

Table 10: Performance evaluation: Differing numbers of virtual robots with a WorldModelExchange sensor

In Table 10 the results of an experiment are shown, where the FUmanoid *SimCase*

was used to simulate robot behavior with differing numbers of virtual robots. Each robot instance was equipped with just one WMExchange sensor. The resulting load and the frame rate of the simulation stepping process were measured. The data shows that the developed simulation software can easily provide six robot instances and simulate a soccer match. A worldmodel of a control software is updated every $250ms$, while the simulation software is still able to provide 84.6 updates per second, even with six virtual robots.

5.3 Motion Control

The use of actuators in a *SimTree* as a virtual humanoid robot was implemented at the end of the practical phase of this diploma thesis. As a result, the motion control was not actively used by other students for their implementation of a dynamic motion algorithm such as walking or kicking a ball.

First experiments showed that the *MotorBus* representation using Unix sockets is a bottle neck, since the timeouts on the physical bus device are smaller and cannot be achieved in the virtual setup. This leads to decreased frame rates for the dynamic algorithms, which has a direct influence on their behavior.

Currently, the robot control software provides a Dynamixel protocol as an adapter interface, which is the origin for the systems difference in their performance. A protocol based on joint angle can simplify the communication process, but is not supported.

A virtual robot motion control was evaluated by generating patterns of target motor positions locally within the simulation software.

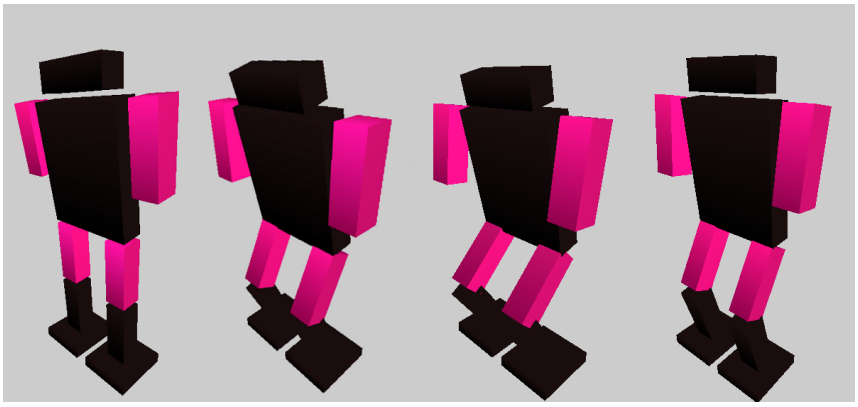


Figure 36: Example of a motion sequence of a virtual robot model

The model shown in Figure 36 was extracted from an RDF and consists of 10 body parts and 9 virtual servomotor devices. It is a simplified version of the full representation of the 2011 FUmanoid robot model and includes only motors manipulating the robots pitch angle. All servomotors are controlled by a PID controller. Although, more complex dynamic motions, such as walking, cannot yet be realised, a simple static knee bend motion can be processed within the simulation software.

Currently, the main problem with dynamic motions is that the motor commands do not reach their recipient in time. This problem was observed in an experiment in which communication with several servomotors of one or more robot instances was attempted as fast and often as possible.

#Robots	#Actuators	Error Rate	Frame Rate	Load
1	1	12.2 %	96.5 fps	25%
1	7	23.1 %	94.6 fps	62%
1	9	30.2 %	91.3 fps	105%
2	1	17.0 %	93.3 fps	75%
2	7	25.2 %	92.8 fps	120%
2	9	36.0 %	90.6 fps	145%

Table 11: Results of the servomotor communication experiment

Table 11 shows the result of this performance measurement test in which robot models which different numbers of servomotors have been compared. The frame rate of the *MotorBus* structure of each robot control software was distinguished as well as the error rate. Therefore, all virtual servomotor devices were connected to a single *SensorSet*. The control software declares all unanswered commands as an error. The resulting error rate describes the ratio between correct transmissions and errors. The load is distinguished by the operating system and has a range from 0 to 200 percent, since the system has two cores.

In summery, the error rate of all tests is very high compared to its common range, when the control software is executed on physical robot hardware. The Unix socket is the bottleneck in this setup's dataflow. It has been observed that exchanging data on the level of the MotorBus Dynamixel protocol is currently not efficient enough to provide the control software with emulated data and to control a dynamic motion.

6 Conclusion

This thesis presents the simulation software of the "FUmanoids" soccer playing humanoid robot team as recently developed for RoboCup 2011.

The task of this diploma thesis was to develop a sensing device emulator for humanoid robots, which would be able to interact with a simulated virtual environment. The intent was for physical feedback on body parts of the robot to be applied by an exchangeable dynamic library. The software had to fulfill all the requirements of existing interfaces in other software packages, such as the robot control software or robot management software. The integration had to be fully transparent, so that none of the other software package would recognize the simulation, but continue to work as if using the real robotic hardware.

Furthermore, the intent was for the outgoing interfaces to be as generic as possible in order to integrate this simulation package into the different software eco-systems of other teams participating in the RoboCup humanoid competitions. The software's design and its integration are often the only issues preventing these software tools being shared between competing teams more often. The exchange of knowledge and achievements of previous development cycles is a key idea of the RoboCup initiative as such. Of course, additional work has to be done to integrate this simulation software into other systems, but this work was reduced to a minimum.

Five different kinds of sensor devices have been implemented in order to reproduce every data stream which would be available on the robot hardware: camera, foot pressure sensor, inertial measurement unit, actuators and a virtual sensor for exchanging the robots world model data. Their functionality and integration into the system have been briefly described.

It has been shown in specific experiments, and by integrating the simulation software into the 2011 development cycle, that the software can have a big impact on important parts of the robot control software development process. The quality can be more easily ensured and evaluated and the projects quality as such can be improved. Its influence on the testing methods for motion control, vision and

localization algorithm evaluation and behavior development has been presented in different scenarios.

In summary, the simulation software in its current version is a preliminary step towards future automated testing possibilities for humanoid robots. An outlook on further improvements and new features for the software is given in the following section.

6.1 Outlook

The evaluation of the experiments and the application of the simulation package during the development cycle of 2011 showed that having a simulation software has a positive effect at all stages of the development process.

The current state of the software still falls short of providing everything needed to improve the work-flow of the development team, although it can be seen as a basic framework and a starting point for further improvements and extensions.

Since the most important benefit provided by this software is its simplification of other work processes, usability and performance can be easily evaluated by examining whether the simulation is used by other developers or not. Whenever test results are inaccurate or do not improve upon the existing testing methods, the module will not be accepted by other developers. This can be seen in how the simulation has changed behavior control development, whereas the benefits did not outweigh the existing drawbacks in the case of the vision module, and the simulation method has not yet replaced legacy methods of vision testing.

A possible future improvement of this software could be the modification of data streams at run time, a very powerful method, which would enrich the current testing methods. Each sensor would need noise filters, which ideally would be configurable by Qt widgets.

Furthermore, the ability to deactivate sensors or attach new ones after a robot has already been initialized would make the entire testing process within the simulator more flexible. In addition, this would remove the need to restart the entire software package.

In the case of the camera sensor, the integration of a reality simulator presented by Yao Fu as a separate camera instance would improve the variety of image types [42]. Since this approach is based on its own image data base storing real images from a robot's camera, it would be a good hybrid approach combining the robustness of the

image database and the flexibility of a freely moving virtual agent in a simulated world. Currently, the conditions for object extracting algorithms are 'too good' in the generated images [42].

Additionally, it is also important to development more lens models. This would help to evaluate the vision under different conditions and to identify which model might be the best before a change is made to the current hardware.

At the moment, a very complex dynamic library is in charge of the kinematic motions of a robot. Especially with many robot instances running expensive sensors such as cameras, it would be an improvement if the motion calculation were not based on a dynamic package, but on a simplified self-implemented approach. In a similar fashion, this could also be tested for collision detection algorithms.

List of Figures

Source Materials

[a] www.robocup.org

[b] http://opende.sourceforge.net/wiki/images/e/ed/Hinge.jpg

[c] Stefan Otte, Entwicklung eines dynamischen Schusses für humanoide Fußballroboter, 2010

[d] Julian Mährlein - www.julianmaehrlein.de

[e] Johannes Kulick, Ein stabiler Gang für humanoide, fußballspielende Roboter, 2011

[f] R. Jonschkowski, Verhaltenssteuerung für autonome human. Fußballroboter mit XABSL, 2010

[g] National Instruments, Inertial Measurement Unit - zone.ni.com/cms/images/devzone/tut/IMU.jpg

All images, which are not labeled with a specific reference or named in the list above[1], have been created by the author of this work and are in public domain.

[1] last updated 23rd January 2012

Bibliography

[1] *Virtual Reality Modeling Language*. http://www.web3d.org/x3d/specifications/#vrml97,

[2] *USARSim: a robot simulator for research and education*. 2007 . – 1400–1405 S.

[3] *Simulation in Encyclopædia Britannica*. 2012

[4] `http://www.airbus.com/innovation/proven-concepts/in-fleet-support/training/`

[5] Airport Research Center: *A380 Studies*. 2012

[6] Aviation Today: *The Innovative Airbus A380*. `http://www.aviationtoday.com/am/categories/bga/206.html`. Version: 2006

[7] Benedetto ; Messina ; Calvi: *Potentialities of Driving Simulator for Engineering Applications to Formula 1*. 2011

[8] Boeing: *Design of a Physics Abstraction Layer for Improving the Validity of Evolved Robot Control Simulations*, University of Western Australia, Diss., 2009

[9] Boeing, Adrian: *Physics Abstraction Layer*. `http://www.adrianboeing.com/pal/`

[10] Boeing, Adrian ; Bräunl, Thomas: Evaluation of real-time physics simulation systems. In: *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. New York, NY, USA : ACM, 2007 (GRAPHITE '07). – ISBN 978–1–59593–912–8, 281–288

[11] Brooks, Rodney A.: Intelligence without reason. In: *COMPUTERS AND THOUGHT, IJCAI-91*, Morgan Kaufmann, 1991, S. 569–595

[12] Coumans, Erwin: *Bullet 2.76 Physics SDK Manual*. 2010

[13] Epic Games: *Unreal engine, www.epicgames.com*. 2005

[14] Fischer ; Heinrich ; Hohl ; Lange ; Langner ; Mielke ; Moballegh ; Otte ; Rojas ; Schmude von ; Seifert ; Steig ; Weissgerber: FUmanoid Team Description Paper 2010. (2010)

[15] Fischer, Bennet: *Hard- und Software-Entwicklung für ein Stereo-Vision-System in eingebetteter Linux-Umgebung*, Technische Universität Berlin, Diplomarbeit, 2009

[16] Friedmann, M. ; Petersen, K. ; Stryk, O. v.: Adequate Motion Simulation and Collision Detection for Soccer Playing Humanoid Robots. In: *Proc. 2nd Workshop on Humanoid Soccer Robots at the 2007 IEEE-RAS Int. Conf. on Humanoid Robots*. Pittsburgh, PA, USA, Nov. 29 - Dec. 1 2007

[17] Friedmann, Martin: *Simulation of Autonomous Robot Teams With Adaptable Levels of Abstraction*, Technische Universität Darmstadt, Diss., Nov. 30 2009. `http://tuprints.ulb.tu-darmstadt.de/2113/`

[18] Friedmann, Martin ; Petersen, Karen ; Stryk, Oskar V. ; Darmstadt, Technische U.: Tailored real-time simulation for teams of humanoid robots. In: *In RoboCup Symposium 2007*, S. 9–10

[19] Fédération Internationale de Football Association: *Laws of the Game 2010/2011*. 2011

[20] Ghazi-Zahedi, T Röfer P Schöll K Spiess A Twickel W. T Laue L. T Laue: *Rosiml - robot simulation markup language*. `http://www.tzi.de/spprobocup/RoSiML.html`. Version: 2005

[21] Google: *Protocol Buffers*. `http://code.google.com/p/protobuf/`

[22] Kahlert, Björn: *Client-Server-Kommunikation für humanoide Roboter*, Freie Universität Berlin, Bachelorarbeit, 2009

[23] Kanda, Takayuki ; Ishiguro, Hiroshi ; Imai, Michita ; Ono, Tetsuo: *Development and Evaluation of Interactive Humanoid Robots*. 2004

[24] Kukulski, Mariusz: *Entwurf und Bau einer humanoiden Bewegungsplattform für Fußball spielende Roboter*, Freie Universität Berlin, Diplomarbeit, 2010

[25] Laue, Tim ; Röfer, Thomas: *SimRobot – Development and Applications* ⋆

[26] Laue, Tim ; Spiess, Kai ; Röfer, Thomas: SimRobot - a general physical robot simulator and its application in robocup. In: *In: RoboCup 2005: Robot Soccer World Cup IX. Lecture Notes in Artificial Intelligence*, Springer, 2006, S. 173–183

[27] Lötzsch, Martin ; Bach, Joscha ; Burkhard, Hans-Dieter ; Jüngel, Matthias: Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. In: *In 7th International Workshop on RoboCup 2003*

(Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence, Springer, 2004, S. 114–124

[28] Michel, Olivier: Cyberbotics Ltd. Webots TM : Professional Mobile Robot Simulation. In: *Int. Journal of Advanced Robotic Systems* 1 (2004), S. 39–42

[29] Moballegh ; Hohl ; Guilbourd ; Oertel ; Rojas: FUmanoid Team Description Paper 2007. (2008)

[30] Nvidia: *PhysX SDK Website*. `http://developer.nvidia.com/physx`. Version: 2012

[31] Nvidia.com: *Qt Online Reference Documentation*. `http://doc.qt.nokia.com`

[32] Petersen, Karen: *Effiziente Kollisionserkennung und echtzeitfähige Simulation der Kinematik, Dynamik und Sensorik autonomer Fahrzeuge*, Technische Universität Darmstadt, Department of Computer Science & Department of Mathematics, Diplomarbeit, 2007

[33] RoboCup Humanoid League Technical Committee: *RoboCup Soccer Humanoid League Rules 2011*. http://www.tzi.de/humanoid, 2011

[34] Robotis: *Dynamixel RX-28 User's Manual V1.10*

[35] Robotis: *Dynamixel RX-64 User's Manual V1.10*

[36] Röfer, Thomas ; Laue, Tim ; Müller, Judith ; Burchardt, Armin ; Damrose, Eric ; Fabisch, Alexander ; Feldpausch, Fynn ; Gillmann, Katharina ; Graf, Colin ; Haas, Thijs J. ; Härtl, Alexander ; Honsel, Daniel ; Kastner, Patrick ; Kastner, Tobias ; Markowsky, Benjamin ; Mester, Michael ; Peter, Jonas ; Riemann, Ole Jan L. ; Ring, Martin ; Sauerland, Wiebke ; Schreck, André ; Sieverdingbeck, Ingo ; Wenk, Felix ; Worch, Jan-Hendrik: *B-Human Team Report and Code Release 2010*. 2010. – 127 pages

[37] Seifert, Daniel: *Portierung der FUmanoids-Software*, Humboldt-Universität zu Berlin, Institut für Informatik, Deutschland, Studienarbeit, August 2009

[38] Seugling, Axel ; Rölin, Martin: *Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool*, Umea University, Sweden, Diplomarbeit, 2006

[39] Smith, R.: *Open Dynamics Engine - ODE*. http://www.ode.org/ode-docs.html, 2005

[40] Tucker, S. G.: Emulation of large systems. In: *Commun. ACM* 8 (1965), December, S. 753–761. – ISSN 0001–0782

[41] von Schmude, Naja: *Selbstlokalisierung humanoider Roboter im RoboCup.* 2011

[42] Yao, Fu ; Moballegh, H. ; Rojas, R. ; Ji, Longxu: Reality Sim: A realistic environment for robot simulation platform of humanoid robot, 2011

[43] Ziegler, J G. ; Nichols, N B.: Optimum Settings for Automatic Controllers. In: *Journal of Dynamic Systems Measurement and Control* (1993), S. 220