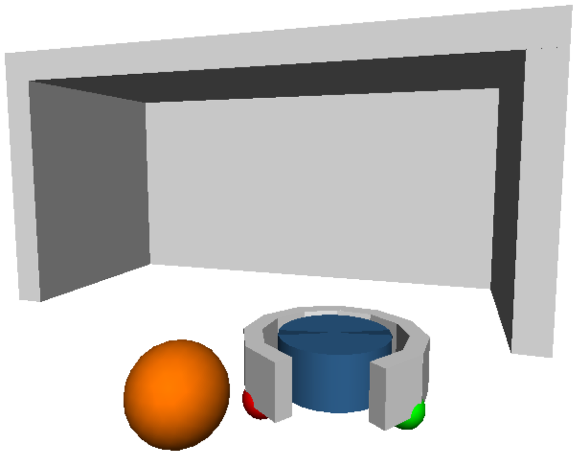
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2010

Eurico Farinha Pedrosa

Ambiente de simulação para agentes em futebol robótico

Simulated environment for robotic soccer agents



Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2010

Eurico Farinha Pedrosa

Ambiente de simulação para agentes em futebol robótico

Simulated environment for robotic soccer agents

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de Prof. Doutor Artur Pereira e Prof. Doutor Nuno Lau, professores auxiliares do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente	**Doutora Maria Beatriz Alves de Sousa Santos** professora associada com agregação da Universidade de Aveiro
vogais	**Doutor Paulo José Cerqueira Gomes da Costa** professor auxiliar da Universidade do Porto
	Doutor Artur José Carneiro Pereira professor auxiliar da Universidade de Aveiro
	Doutor José Nuno Panelas Nunes Lau professor auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

Quero começar por agradecer aos meu pais, Luisa e Fernando, pelo sempre presente apoio que me deram ao longo destes anos e que não será esquecido. Agradeço também à minha madrinha Otilia pelo seu apoio e pela chávena de café que está sempre à minha espera.

Aos professores Nuno Lau e Artur Pereira agradeço pela orientação que me deram e pelas sempre interessantes discussões que se levantaram durante o desenvolvimento desta dissertação. Agradeço a todos os elementos da CAMBADA pela paciência que eu os interrompia com as minhas dúvidas.
Por fim agradeço a todos os meus amigos por tão bons momentos passados. . .

palavras-chave Simulação, Robótica, RoboCup, Sistemas multi-agente, Modelação

Resumo

O teste de algoritmos na área da robótica pode ser uma tarefa difícil, especialmente se o teste envolver múltipos robots. Neste contexto o uso de um simulador torna-se uma ferramenta importante no teste de algoritmos pois permite ultrapassar algumas limitações e oferece várias vantagens.

CAMBADA é a equipa de futebol robótico da liga de tamanho médio da Universidade de Aveiro, Portugal. A equipa está familiarizada com as limitações do uso de robots reais para o teste de algoritmos. Devido a isso o simulador criado pela equipa Brainstormers Tribots foi adaptado para prover um ambiente de simulação ao software CAMBADA e estava em uso aquando do início desta dissertação. O simulador oferecia pouca flexibilidade na modelação dos robots que resultava em comportamentos imprecisos, oferecia também reduzida interacção com a simulação.

O objectivo desta dissertação é criar um ambiente de simulação para agentes em futebol robótico com a intenção de melhorar o ambiente de simulação da equipa CAMBADA. O simulador deve ser capaz de simular dinâmica de objectos a três dimensões, sensores e actuadores ao mesmo tempo que oferece visualização do mundo e a possibilidade de interagir com a simulação.

Da pesquisa realizada sobre simuladores robóticos o simulador Gazebo respeitava os nossos requisitos e foi escolhido para código base do nosso simulador. Para criar um ambiente simulado adequado à equipa CAMBADA alguns componentes do Gazebo foram alterados e novos sensores e actuadores virtuais foram desenvolvidos. Vários componentes do software CAMBADA tiveram que sofrer alterações de modo a suportar um ambiente simulado. O robot virtual foi modelado de modo a assemelhar-se com o robot real com o objectivo de obter comportamentos mais precisos.

O simulador desenvolvido substituiu a solução anteriormente criada pela equipa CAMBADA e foi usado nos testes de preparação para a participação da equipa no RoboCup 2010 em Singapura onde deu o seu contributo na obtenção do terceiro lugar

keywords Simulation, Robotics, RoboCup, Multi-agent systems, Modeling

Abstract

In the field of robotics, testing algorithms with the real robots can be a difficult task, specially if the test involves more than one robot. In this context a simulator is an important tool for testing algorithms because it helps overcome some limitation and offers several advantages.

CAMBADA is the RoboCup MSL soccer team of the University of Aveiro, Portugal. The team is familiar with the limitations of using the real robots for testing algorithms. Therefore, a simulator created by the Brainstormers Tribots team was adapted to provide a simulated environment for their software and was used for testing at the time of the beginning of this thesis. The simulator offered low flexibility on the modeling of the robots from which resulted inaccurate behaviors, it also offered reduced interaction with the simulation.

The purpose of this thesis is to create a simulation environment for robotic soccer agents with the intention of improving the simulated environment for the CAMBADA team. The simulation must provide three-dimensional dynamics of objects, be capable of simulating sensors and actuators, allow the visualization of the simulation and provide interaction with the simulation.

From the conducted survey about robotic simulators, the simulator Gazebo complied with our requirements and was chosen to provide the code base for our simulator. To create an adequate simulation environment for the CAMBADA team some components of Gazebo were modified and new sensors and actuator were developed. Several components of the CAMBADA software had to be modified to support the simulated environment. The virtual robot was modeled to resemble the real robot to provide more accurate behaviors.

The developed simulator substituted the previous solution created by CAMBADA team and was used in the preparation tests for the participation in the RoboCup 2010 in Singapore where it contributed to obtain of the third-place.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The software of a robotic agent has in its blocks a large variety of algorithms, each one with a specific goal. While developing these algorithms, testing is a mandatory task so that its results can be observed, analyzed and discussed. The ideal approach for testing is to use the real robot, but this approach can sometimes be problematic, specially if more than one robot is needed. Using real robots for testing is not without constraints. Sometimes there may not be enough robots available (either by technical issues or by monetary issues), they can be damaged, the physical space required for testing is bigger than the available, the programmer can be physically distant from the robots, etc. In this context, using a simulated environment makes perfect sense as it allows to overcome these imposed constraints.

By using a simulated environment the real robots are no longer necessary and, as result, several advantages arise [1]:

- Less expensive than the real robots;
- No damage is made to the real robots while testing (they are not used);
- Save the robots batteries;
- Less testing time;
- Easy testing of new algorithms;
- Easy development and testing of new models of robots;
- Different components can be easily added or removed for testing (even the non-existing ones);
- Several levels of abstraction can be studied;
- More detailed information can be extracted from the environment;

- Facilitates study of multi-agent coordination methods;
- Control over the simulation time.

Even though a simulated environment gives good advantages for development, it does not always offer a satisfactory transfer to reality [2]. Therefore, efforts are needed to create more accurate simulated models and trustworthy behaviors.

1.2 Objectives

The aim of this thesis is to tackle the problem of using real robots in the development and testing of algorithms. The main goal is to create or find a suitable virtual environment for the CAMBADA team to aid development. The specific goals of this thesis include:

- Study the existing solutions for robotic simulation;
- Develop or adapt a robotic simulator;
- Model all entities that have a role in the game (e.g. goals, ball, robots);
- Develop the virtual version of the robot sensors and controllers (e.g. vision, kicker);
- Develop an intuitive interaction with the simulator.

1.3 Requirements

In order to have a more accurate simulation model and reliable behaviors the proposed simulated environment must comply with a set of requirements. They falls in two categories, software requirements and operational requirements.

1.3.1 Software requirements

Rigid Bodies Dynamics
The simulation must be able to detect collisions of rigid bodies, apply joint constraints (e.g. wheel joints) and follow the laws of classic physics.

Multiple Entities
An entity can be a lot of things, a goal, a robot or even only a simple geometry. The simulator must be flexible enough to allow multiple instances of multiple entities.

Sensors/Controllers
There must be the notion of sensor and controller with more than one level of abstraction (e.g. noise, delay). The virtual sensor must be able to obtain data from the simulated environment while the virtual controllers should be able to interact with

virtual sensors, joints or entities in the simulated world. The virtual sensors and controllers can be attached to a physical entity (e.g. robot body) or, to an empty entity creating the concept of them being “on the air”.

Flexible models

By using a configuration file (e.g. xml), the simulator must be able to allow the building of complex models by smaller components like geometries, sensors and controllers.

1.3.2 Operational requirements

Open Source

In the spirit of sharing knowledge an Open Source simulator is a welcome requirement. Also, developing a robot simulator from scratch may not be feasible, therefore, an Open Source solution has the advantage of offering a product free of charge and supported by a community of users and developers.

Unix

The CAMBADA software is developed to work on a unix system namely Ubuntu [1], therefore the simulator must also run in the same system.

Visualization and interaction

The simulator should provide a graphical visualization of the simulated world and allow interaction with it, but this should not be a requirement for the simulation to run. The visualization should be able to run on systems with low graphic capabilities.

CAMBADA agent integration

The virtual sensors and controllers of the simulator to be developed must work intrinsically with the CAMBADA's robotic agent. Therefore the agent should not be aware in which environment it is operating while maintaining the original software architecture and workflow. Also, it is desirable to have all agents running in the same computer.

1.4 Thesis outline

This thesis is organized in five more chapters.

Chapter 2 gives an introduction about robotic simulation and presents a survey of several robotic simulators. In the end, a brief comparison between the described robotic simulators is presented.

Chapter 3 introduces the CAMBADA team and the RoboCup project. Afterwards, the general architecture of the CAMBADA robots is presented followed by the CAMBADA hardware and software.

[1] `http://www.ubuntu.com`

Chapter 4 describes the development of the simulation environment for the CAMBADA team. It starts with the understanding of the simulation environment, presents the modeling of physical entities within the simulated environment, discusses the created virtual sensors and controllers, and finishes with the visualization and interaction provided by the developed simulator.

Chapter 5 presents the results obtained by the developed simulation environment. It focuses in the motion of ball, the developed virtual sensors and controllers, and in the execution times of the simulation.

Finally, chapter 6 discusses the obtain results and presents suggestion for future work.

Chapter 2

Robotic Simulation

2.1 Introduction

A robotic simulator is a tool that provides a quicker and/or simpler way for testing out ideas, theories and software with robots without depending physically on the actual machine, thus saving time and money. A simulation can be used for testing new ideas without having to spend time and money building the robot to do it. The task of testing algorithms for robots may be difficult to accomplish. Turn on the robot, program it, place it in position to start, let it run and record what happened. Given the results this processes is repeated a few times while tweaking the algorithms. Each run may take some time to complete, not to mention the recharging of batteries and the failure of hardware. With a simulator, test algorithms can be accomplished without the time overhead of dealing with the real robot. In addition, while designing a robot we can test different configurations and determine which one provides better performance to our robot without having to build and test a new robot for each configuration [3].

A robotic simulator is a software capable of modeling robots and its environment. The *robotic simulation* has a virtual version of one or more robots and it is capable of emulating the behavior of the real robots in their working environment. The visualization component of a robotic simulator has the purpose of visually demonstrating the simulation and can provide direct interaction with the simulation state. There are several robotic simulators available and the differences between them are the features they provide, such as robot prototyping, physics engine, three-dimensional rendering, scripting, etc. The simulators can target a specific type of simulation environment or, be completely generic allowing the user to create their own simulation environment.

This thesis focuses on creating a simulation environment for robotic soccer agents in the context of a RoboCup MSL team. Therefore we conducted a survey of some of the existing robotic simulators so we could find the adequate tool for our purposes. Our points of analysis were the structure and capabilities of the software, how the simulated world is represented (i.e. how it is modeled), the simulation environment, the visualization component and the interaction with the simulation. We also described the physics engine used by most of the robotic simulators covered by the survey.

The described robotic simulators that follow were chosen by their application on the RoboCup domain or due to their high visibility in the robotic simulation world.

2.2 Physics engine

A common trait between robotic simulators is the use of a physics engine to provide real-time realistic dynamics. They can provide rigid and soft body dynamics (with collision detection), and fluid dynamics. There are several engines available, such PhysX[1], Havok[2], Newton Game Dynamics[3], Bullet[4], Open Dynamics Engine (ODE), etc. The physics engine ODE is commonly used by robotic simulators. Therefore, in this section, we will present some of the engine characteristics. The description is based on the ODE manual [4].

2.2.1 Short introduction

Open Dynamics Engine is a free, high quality library for simulating articulated rigid-bodies. An articulated structure is created when rigid bodies of various shapes are connected with joint of various types. Some of the key feature of ODE are:

- Rigid bodies with arbitrary mass distribution.
- Joint types: ball-and-socket, hinge, slider (prismatic), hinge-2, fixed, angular motor, linear motor, universal.
- Collision primitives: sphere, box, cylinder, capsule, plane, ray, and triangular mesh, convex.
- Collision spaces: Quad tree, hash space, and simple.
- Contact and friction model.
- Has a native C interface (even though ODE is mostly written in C++).

2.2.2 Rigid bodies

A rigid body (Figure 2.1) has several properties that changes during the simulation, such as: the position vector (x, y, z) of the body's reference point, which correspond to the body's center of mass; the linear velocity of the point of reference; the orientation of the body represented by quaternions; the angular velocity of the body. Other properties are

[1]`http://developer.nvidia.com/object/physx.html`
[2]`http://www.havok.com/`
[3]`http://newtondynamics.com`
[4]`http://bulletphysics.org`

constant, including the mass of the body, the position of the center of mass with respect to the point of reference, and the inertia matrix.

A shape of a body is not a dynamical property. It is only *collision detection* that cares about the shape of the body. ODE provides a wide variety of geometry classes, including sphere, box, cylinder, plane, ray, convex, etc.

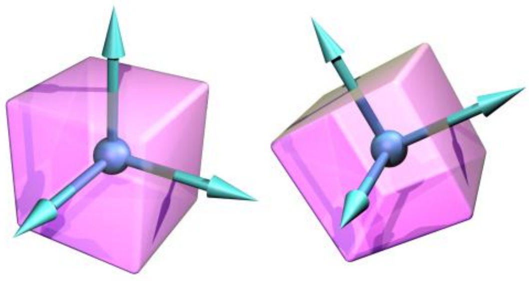


Figure 2.1: ODE rigid body illustration.

2.2.3 Joints

In ODE, much like in real life, joints are used to connect two object. They enforce a relationship between two bodies so they can only have certain positions and orientations to each other. This relationship is called a *constraint.* The Figure 2.2 shows three different constraints. The first is a ball-and-socket joint that constraints the "ball of one body to be in the same location as the socket of another body". The second is a hinge joint and constraints the two parts of the hinge to be in the same location and lineup along the hinge axis. The third is a slider joint that constraints the "piston" and the "socket" to lineup while constraining both bodies to maintain the same orientation.

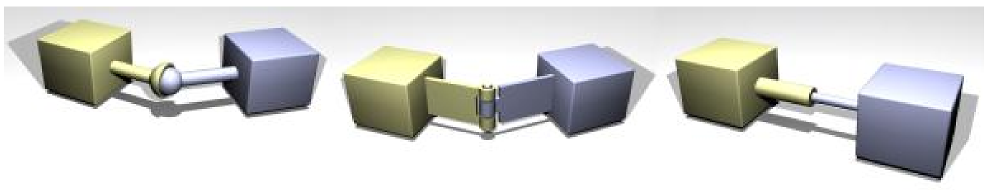


Figure 2.2: Three different constraints types, from [4].

When two bodies are connected by joint, those bodies are required to have certain positions and orientations relative to each other. However, sometime the bodies can be in positions and orientations that violates the constraint. This can happen if the user sets the position and/or orientation of a body and does not set properly the position and/or orientation of the other body, or the accumulation of error during the simulation can drift the bodies from their required positions.

To reduce joint error, at each simulation step each joint applies a special force to bring the bodies to the correct alignment. This force is controlled by the error reduction parameter (ERP), which has value between 0 and 1 that specifies what portion of the joint error will be fixed at the next simulation step.

Constraints are typically “hard”, i.e. constraints represent conditions that are never violated. Although, in practice constraints can be violated by unintentional introduction of errors by the system, but the ERP can be used to correct these errors. Not all constraints have to be “hard”, some “soft” constraints are designed to be violated. Tipicaly, the contact joint acts like a metal that prevents objects from penetrating upon collision. A soft constraint can be used to simulate softer materials, thereby allowing some penetration of the two objects when they are forced together. The distinction between hard and soft constraints are control by two parameter, the ERP, already introduced, and the constraint force mixing (CFM). ERP and CFM can be independently set in many joints.

2.2.4 Collision handling

Before each simulation step, the collision detection function is called by the user to determine what is touching what. These functions returns a list of contact points that specifies a position in space, a surface normal vector, and penetration depth. A special contact joint is created for each contact point. The contact joint receives extra information about the contact, such as friction, how bouncy or soft it is, etc. Then, the simulation step takes place, afterward all contact joints are removed from the system. A complete description of the physics parameters can be found in [4].

2.3 Brainstormers Tribots

Brainstormers Tribots [5] are a RoboCup MSL soccer team founded in 2002. In 2005 after the German Open[5] they released a source code package with software used by their team. The package contained a robotic simulator developed specifically for their MSL robots.

2.3.1 Software overview

The simulator is developed in the C++ programming language for the Linux operating system. For physics it uses the library ODE [6] that simulates articulated *Rigid Body Dinamics*. The software has an initiation stage and then goes into a sequential cycle with fixed time delay at the end, its workflow is depicted in Figure 2.3.

[5]`http://www.robocup-german-open.de`

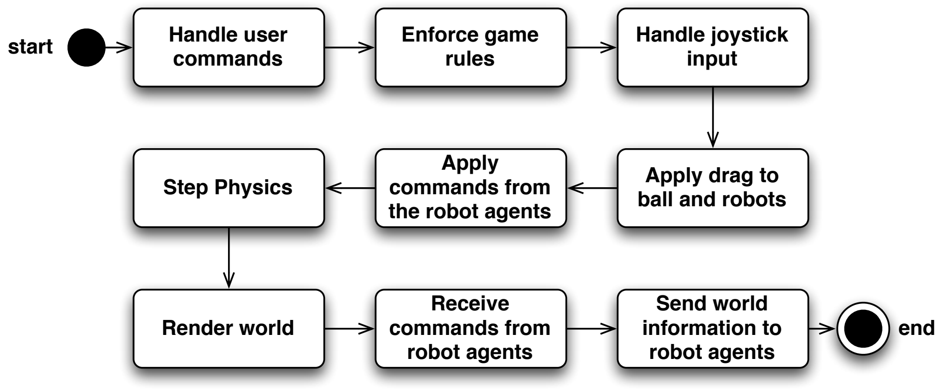

start
Handle user commands
Enforce game rules
Handle joystick input
Apply drag to ball and robots
Apply commands from the robot agents
Step Physics
Render world
Receive commands from robot agents
Send world information to robot agents
end

Figure 2.3: Brainstormers Tribots simulator workflow at every cycle.

2.3.2 World representation

The simulator created by the Brainstormers Tribots team is a RoboCup MSL centric simulator, meaning it was developed specifically for the MSL soccer game scenario. It models a field, two goals, a ball and several robots. Each one of which modeled with a physical body and respective shapes, handled by the ODE library for *Rigid Bodies Dynamics* simulation. All mentioned models are hardcoded into the source code.

One characteristic of the models is that they are very simple, especially the robot model – its a box without wheels. The game field is a plane, representing the whole ground, with its game area limited by a set of line segments. A goal is built by boxes positioned to recreate the posts, the crossbar and the net. Finally, the ball is a sphere. The models can seen in Figure 2.4.

2.3.3 Simulation environment overview

The simulation environment is populated by the players of a MSL soccer game. The game field and the goals are static but the robots are controlled by robotic agents. Rules of the game are checked and forceed at each cycle. Goals, throw-in's and corner are enforced by repositioning the game ball to the appropriate location without changing the play mode.

Playing in a environment ruled by the laws of physics, the motion of the robots is the result of forces directly applied to the robot body (a box). The force values are determined by the motion commands sent by the robotic agent. There is also a kick, which is modeled by a velocity applied to the ball whenever a kick command is received either by the robotic agent or by the user input.

The communication with the robotic agents is made through UDP sockets. Each time communications are issued the simulator checks for incoming commands and saves them in the respective robot state structure, then it sends the state values of each robot to the respective robotic agent. The sent robot state values, such as position, velocity and orientation are gathered from the simulated physics world.

2.3.4 Visualization and interaction

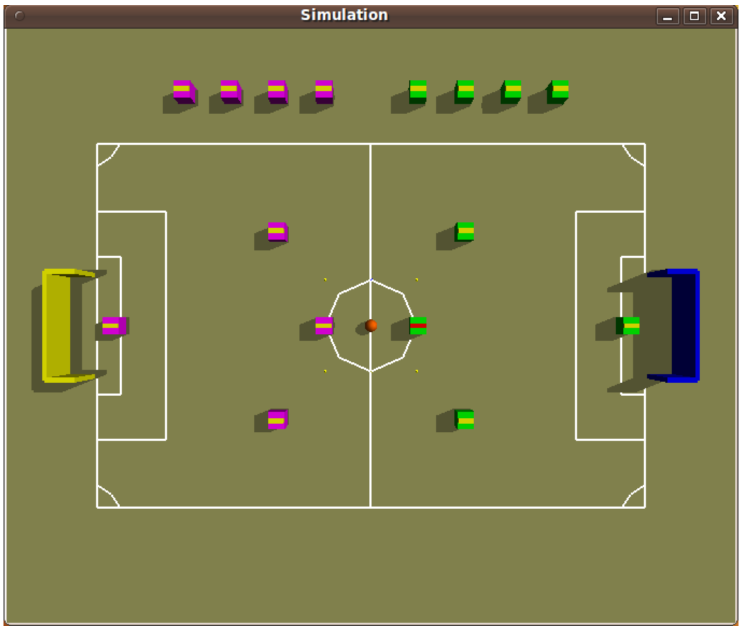


Figure 2.4: Brainstormers Tribots simulation rendering and interaction window.

For visualization purposes is uses the library *drawstuff*, part of the ODE package, that is capable of rendering a simple virtual 3D world (Figure 2.4). The simulator renders a representation of its world at each cycle, it draws the field, the goal and the robots models. What is rendered for the models is the 3D representation of their bodies geometric shapes. The field has no geometry because it is modeled as a plane, nevertheless it is rendered as a big quadrangle with a texture of a grass field.

Besides the common interaction with the rendering view, the simulator also provides interaction with the simulation state. It is possible to control the virtual ball and virtual robots with the keyboard or with a joystick, record the world state and/or save it to a file, and recall the world state from a file.

2.3.5 CAMBADA simulator

CAMBADA[6] are RoboCup MSL soccer team of the University of Aveiro, Portugal. Before the beginning of this survey the CAMBADA team used a modified version of the simulator made available by the Brainstormers Tribots team (Section 2.3). The simulator was adapted to create a simulated environment for the CAMBADA software. The result is a software with the same characteristics of the Brainstormers Tribots simulator but with different requirements, shown in Figure 2.5.

[6]http://www.ieeta.pt/atri/cambada/

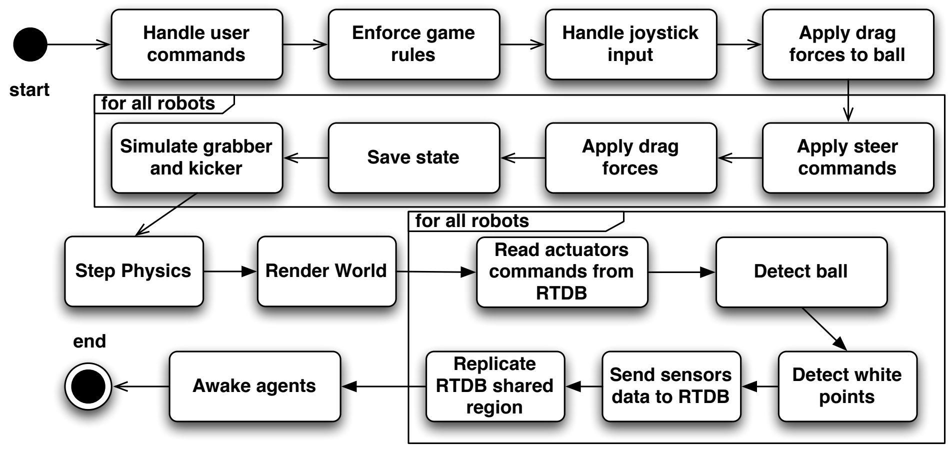


Figure 2.5: CAMBADA simulator workflow at every cycle.

Also, the simulator in the initiation stage loads the field dimensions from the CAMBADA configuration file, initiates the Process Manager that manages the agent processes, and also initiates the RTDB for all agents.

2.4 RoboCup Simulator Dev

The *RoboCup Simulator Dev* is a RoboCup MSL simulator developed and released as open source by the Hibikino-Musashi Team [7].

2.4.1 Software overview

The simulator is developed in C++ for the Microsoft® Windows® platform. It has a three-dimensional rigid bodies dynamics simulation provided by the library ODE [6]. For rendering purposes it uses the library *drawstuff* that is part of ODE. The simulator has a single thread of execution divided in three stages: *initiation*, then *simulation loop* and finaly *cleanup*.

The *initiation* stage has to load the simulation configuration file, load the user input devices, configure visualization engine and finaly build the physics world.

The *simulation loop* (Figure 2.6), like the name implies, is where the simulation takes place until the user quits and is handled by the library *drawstuff*. The simulation loop tries to synchronize its simulation with the realtime time by adapting the physics time step to the rendering frame rate.

The *cleanup* stage safely releases the resources allocated for the simulation.

Despite of being a robotic simulator it does not provide any form of communication with external robotic agents, but it has a cooperative decision and behavior module that can control the robots. The module gathers sensorial data from all its team mates to

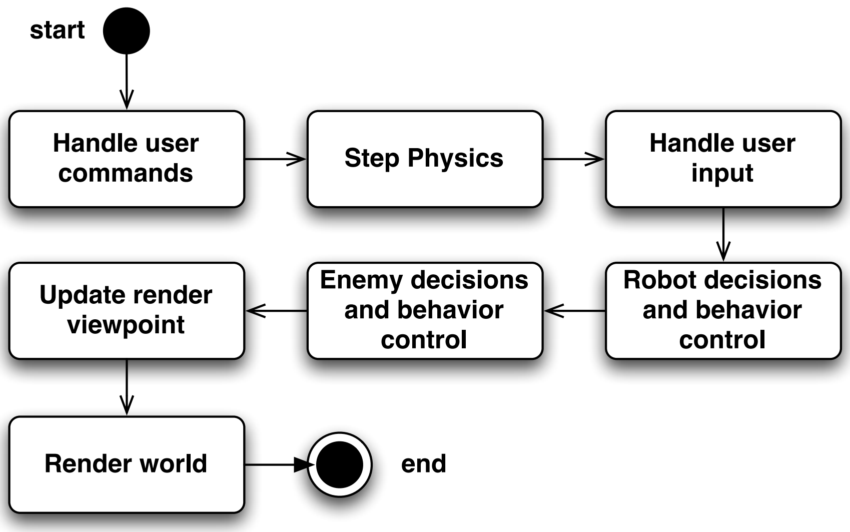


Figure 2.6: RoboCup Simulator Dev simulator workflow at every cycle

determine its decisions and respective behaviors.

2.4.2 World representation

Being a RoboCup MSL simulator it models a field, goals, a ball and robots. All entities in the world are modeled with a physical body and respective shapes, handled by the ODE library. The models are defined directly in the source code but it has the ability to create several instance of the same model (e.g. robot). The number of robots present in the simulation is defined in the configuration file. The game field is a plane, representing the whole ground, with a texture of a soccer grass field and delimited by boxes around the game area. The goal is built by boxes positioned to recreate the posts, the crossbar and the net. Finally, the ball is a simple sphere. Because the simulator was developed to serve the Hibikino-Musashi Team requirements the robot models resembles their real robot in terms of shape and motion (holonomic motion). It has its complete chassis and also the holonomic motion setup. The wheels of the robot are modeled as cylinders and connect to the robot chassis by hinge joints. The physics parameters for geometry contact, such as friction and slip, are hardcoded.

2.4.3 Simulation environment overview

The simulation environment is populated by the players of a MSL soccer game. There are two teams, one is the team controlled by the user and the other is the opponent team. The opponent team is controlled by a cooperative decision and behavior module trying to win the game. The team controlled by the user is actually also controlled by the cooperative decision and behavior module except one robot that can be controlled by the user.

In terms of motion, the holonomic motion behavior is achieved by applying the kinematic model to the robot. The desired linear and angular velocities of the robot frame is translated to the angular velocities of the wheels and applied to the hinge joints. The desired behavior is achieved by fine tuning of the physics parameter for surface contact.

2.4.4 Visualization and interaction

For visualization purposes is uses the library *drawstuff*, part of the ODE package, that is capable of rendering a simple virtual 3D world (Figure 2.7) . The simulator renders a representation of its world at each cycle, it draws the field, the goal and the robots models. What is rendered for the models is the 3D representation of their bodies geometric shapes. It also renders text for information purpose, such as when the simulation is pause or when a team scores.

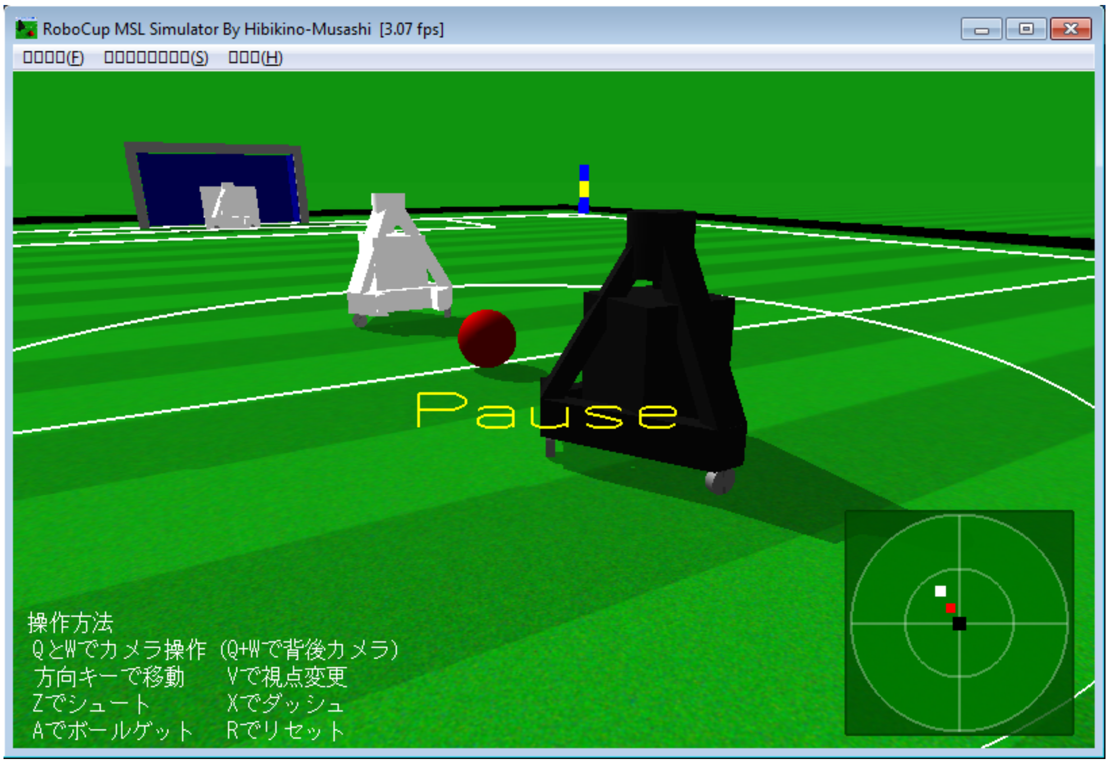

RoboCup MSL Simulator By Hibikino-Musashi [3.07 fps]
□□□□(F) □□□□□□□□(S) □□□(H)
Pause
操作方法
QとWでカメラ操作 (Q+Wで背後カメラ)
方向キーで移動 Vで視点変更
Zでシュート Xでダッシュ
Aでボールゲット Rでリセット

Figure 2.7: *RoboCup Simulation Dev* rendering and interaction window.

Beside the common interaction with the rendering view, the simulator also provides interaction with the simulation state. It it possible to control a robot with the keyboard, joystick or Wii[7] controller. It is possible to reset the simulation, it brings all robots and ball to their initial state.

[7]`http://pt.wii.com/`

2.5 SimSrv - A RoboCup F2000 Simulator

The SimSrv [8] is a simulator for the RoboCup MSL, a joint open source project by the universities of Freiburg [9] and Stuttgart [10] that presents a flexible simulation platform instead of a rather specialized for a certain robot architecture and software.

2.5.1 Software overview

The simulator is developed in C++ and based on server/client concept with plugin architecture to provide maximum modularity. The server, at its core, is a 2D physics simulation where the world state is uninterruptedly updated within discrete time intervals δ_t. At time t the following world state for time $t + \delta_t$ is calculated taking into account the object velocities and changes to internal state. The physics simulation is basically influenced by the the robot objects existing in the simulation (Figure 2.8). The implementation of the physics simulation is based on the book "Dynamics" [11].

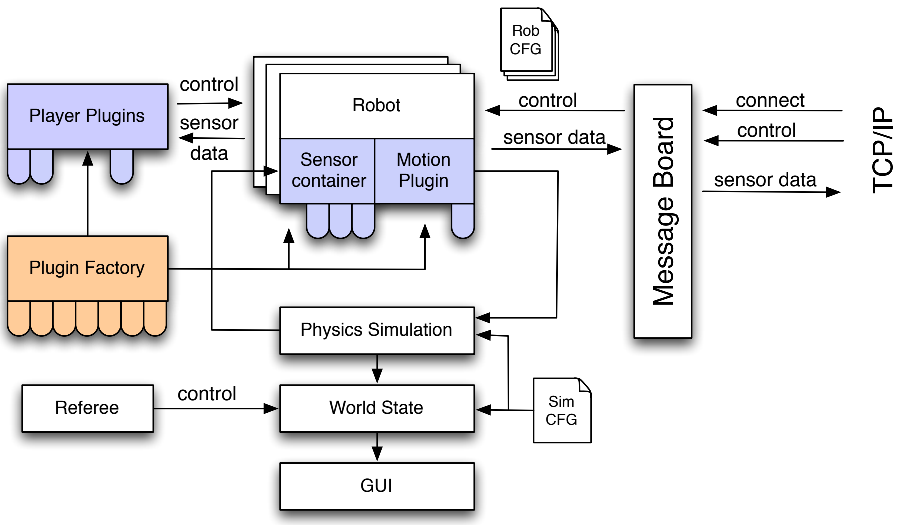


Figure 2.8: Overview of the server architecture, adapted from [8].

Robots can be controlled by clients connected to the *Message Board* via TCP/IP socket or by a *player plugin* created in the GUI. Using TCP/IP socket provides the advantage that a client can be developed in any kind of programming language as long they follow the protocol. For simplicity, the protocol is based on ASCII strings that start with a command keyword followed by a list of parameters. Alongside the commands that controls a robot, the protocol also defines commands to control the simulation (e.g. reset the simulation). Due to the generic plugin architecture there is no formal definition for the exchange of sensor data, the plugin itself is responsible to pack and unpack the data. Because the server and client usually run on a different machine the protocol has a time synchronization mechanism.

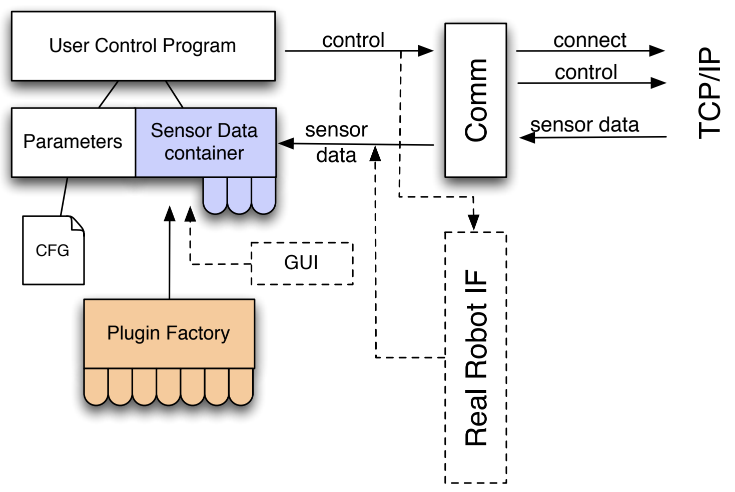


Figure 2.9: Overview of the client architecture, adapted from [8]. Dashed components are optional.

The SimSrv also provides a client architecture, depicted in Figure 2.9. The client is built up by a module that handles communication with the server (*Comm*), a module for parsing the robot configuration file *Parameters* and a *sensor container* that holds the sensor plugins created by the *plugin factory*. The robot's configuration file holds information about the sensors and maximum velocity parameters. When connecting with the server the configuration file is transmitted to the server and used there to generate an equivalent robot object. The sensor plugins created on the client are equal to the ones created on the server but with reduced capability of producing sensor data.

2.5.2 World Representation

The world of the SimSrv is flat, i.e. the world has a 2D representation and is ruled by a 2D physics engine. Although if offers a limited support for the third dimension when required (e.g. three-dimensional sensor information). From the simulation configuration file, the environment is defined by geometric primitives such as rectangles, ellipses and triangles allowing to build almost any 2D scenario. Because the SimSrv is a simulator for the RoboCup MSL it provides representations of the game field and ball, both configurable in the simulation configuration file.

Each robot has a *motion plugin* and *sensor container*. The motion plugin allows the simulation of the robot dynamics. The sensor container has the various simulated sensors. The body of the robot is defined by the same geometric primitives that builds up the environment, allowing the simulation to provide rigid body collisions. Manipulators are variable parts of the robot geometry, they can have several states were each of which represents a different geometric configuration. Also, manipulators can have a parameter that represents their ability to accelerate the ball. The single configuration of a robot is defined by a file.

To help the creation of new scenarios and robots the SimSrv package provides a *What You See Is What You Get* designer, shown in Figure 2.10.

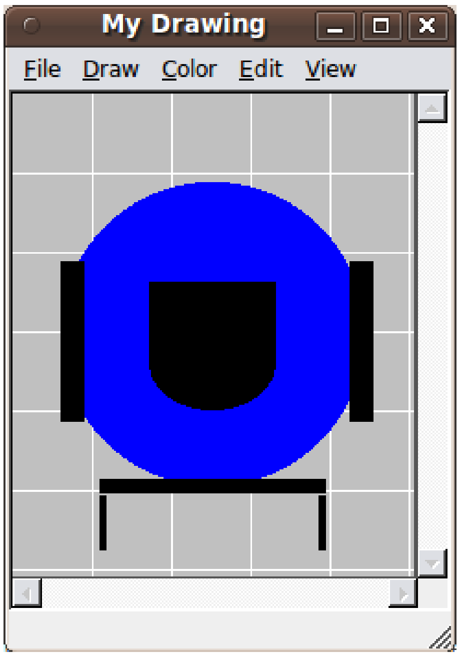


Figure 2.10: SimSrv scenario and robot designer.

2.5.3 Simulation environment overview

The defined environment is populated by robots, each of which controlled by a client (e.g robotic agent) or by a *player plugin*. There is a referee entity intended to enforce the rules of a MSL soccer game, although it only keeps the score updated, resets the game when someone scores and brings the ball into the game if it crosses the game field boundary.

The *motion plugin* within the robot offers different kinematic models, such as differential drive or holonomic motion (three wheels only). The client can control the robot by sending motion commands to the *motion plugin*, to address the different types the standard commands `setRotationalVelocity` and `setTranslationalVelocity` are sent, but for more complex models a n-dimensional vector can be used. A motion model may need to send its internal state back to the client, for that the default odometry sensor of the server is invoked making the data available on both sides.

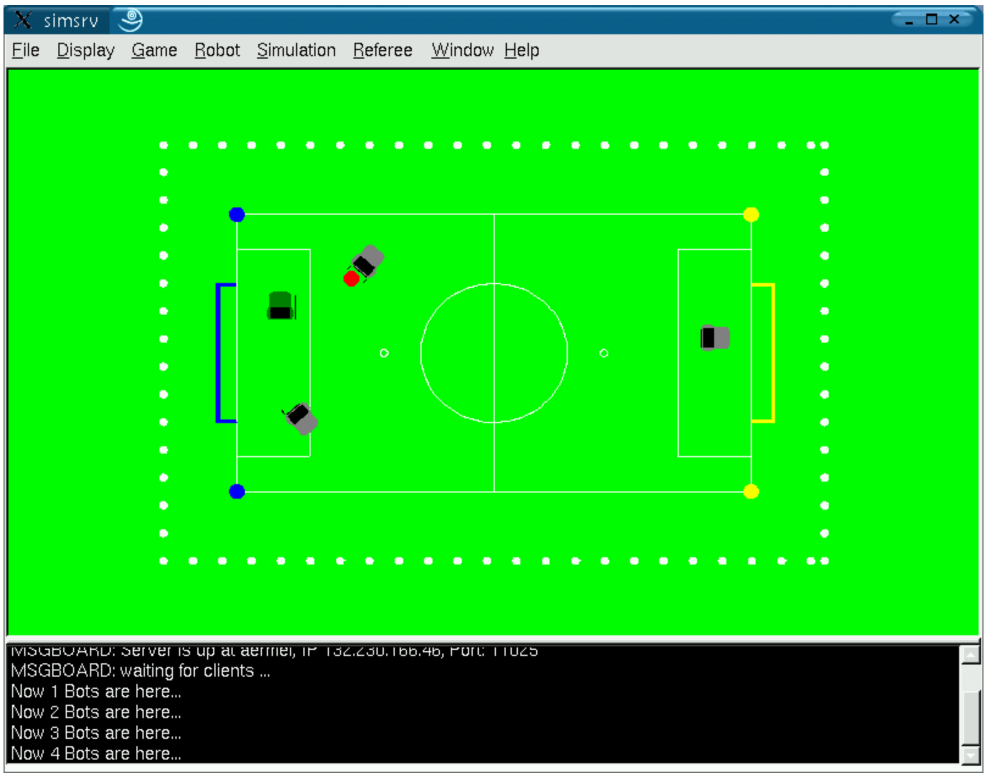
The sensors data is only sent to the client when requested. The data is packed by the sensor plugin and then sent to the client, there the corresponding sensor plugin unpacks the data. The simulator offers sensors plugins for odometry, camera for ionic data of ball, goal and corner posts (considering a maximum view distance), ionic omnivision camera, laser range finder, ultrasonic and camera image by 3D scene reconstruction.

The *player plugin* is an interesting concept. Instead of having a full fledge robotic agent controlling the robot to execute a simple behavior (i.e. opponent goalkeeper) we can have a plugin doing just that but without the overhead of a remote client. It can communicate with other *player plugins* for cooperative behaviors and it can also manipulate the world state to overcome some limitations created by the virtual sensors.

2.5.4 Visualization and interaction

The SimSrv has a GUI built in Qt[8] (Figure 2.11) that serves two purposes, display the simulation environment and interact with the simulation. The GUI draws the two-dimensional environment of the simulation from a top view, also it can show information about the world state (e.g. robot) and *Message Board* messages.

The GUI can influence the simulation by directly manipulating the state of the object present in the simulation, such as positioning the the ball and manipulate its velocity using the mouse or controlling a robot with the keyboard. In realtime it can add new *player plugins* to the simulation. It can also manipulate the simulation indirectly by controlling the *referee* module. Finally, the simulation time rate can also be controlled by the GUI.



```
MSGBOARD: Server is up at aermel, IP 132.230.166.46, Port: 11025
MSGBOARD: waiting for clients ...
Now 1 Bots are here...
Now 2 Bots are here...
Now 3 Bots are here...
Now 4 Bots are here...
```

Figure 2.11: SimSrv Graphical User Interface, from [12]

[8] http://qt.nokia.com/

2.6 Webots$^{\text{TM}}$

Webots$^{\text{TM}}$ [13] is a commercial software for mobile robotic prototyping simulation and transfer to real robots. Created by Cyberbotics Ltd.[9] Webots$^{\text{TM}}$ is intended for researchers and teachers in the area of mobile robots.

2.6.1 Software overview

Webots$^{\text{TM}}$ can provide modeling for any mobile robot, such as wheeled, legged and even flying robots. It runs on Windows, Linux and Mac OS X. It includes a library of sensors that the user can include in the model and fine tune individually (e.g. noise, range). Sensors such as distance sensor, range finders, light sensor, touch sensor, global positioning sensor and positions sensors for servos are a few examples of available sensors in the library. A library of actuator also exist, for example servos, LEDs and grippers are available.

Complex environment can be created with the use hardware accelerated OpenGL technologies including fog, texture mapping, shading, etc. Three-dimensional models can be imported through the VRML97 [14] standard. The robots and the environment are defined by a *world* file.

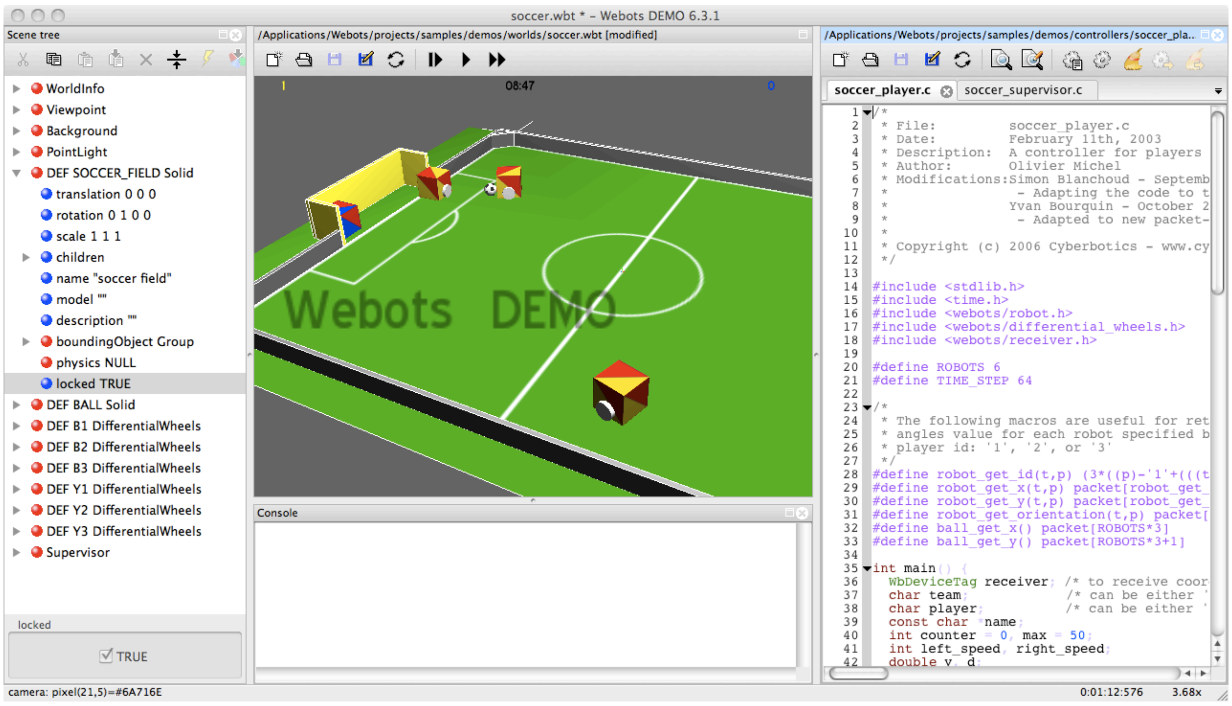
The use of virtual time by the simulation system allows simulations to run much faster that it would on the real robot. The simulation physics relies on ODE to perform accurate simulation whenever is needed. Each component of a robot is bound to an object for physics collision and has associated parameters that influence the physics behavior (e.g friction coefficients, bounciness).

Programming the robots can be done in C, C++ and Java, or from third party software through TCP/IP. Webots$^{\text{TM}}$ has built-in editor with syntax highlight and auto-completion for Webots Application Programming Interface (API). The source code can be compiled and executed in the simulation. In fact, Webots$^{\text{TM}}$ provides a full fledge development environment [15], shown in Figure 2.12.

2.6.2 World representation

A world in Webots$^{\text{TM}}$ is represented as a scene tree that describes the environment, the robots and its graphical representation. The scene tree is structured like VRML97 [14] file. The tree is composed by nodes, a node can be a field containing values (numerical value, text strings) or other nodes. A user can create or modify a world by editing the scene tree using the *Scene Tree* window (Figure 2.12). While editing the scene tree, its changes immediately affect the simulation as can be seen in the 3D window (Figure 2.12). Also the 3D window can be used to directly manipulate the location and rotation of a node. For more information on how to model a world in Webots$^{\text{TM}}$ please refer to [16] .

[9] `http://www.cyberbotics.com`


soccer.wbt * – Webots DEMO 6.3.1
Scene tree
/Applications/Webots/projects/samples/demos/worlds/soccer.wbt [modified]
/Applications/Webots/projects/samples/demos/controllers/soccer_pla..
08:47
Webots DEMO
Console
soccer_player.c
soccer_supervisor.c
locked
TRUE
camera: pixel(21,5)=#6A716E
0:01:12:576
3.68x

```
WorldInfo
Viewpoint
Background
PointLight
DEF SOCCER_FIELD Solid
  translation 0 0 0
  rotation 0 1 0 0
  scale 1 1 1
  children
  name "soccer field"
  model ""
  description ""
  boundingObject Group
  physics NULL
  locked TRUE
DEF BALL Solid
DEF B1 DifferentialWheels
DEF B2 DifferentialWheels
DEF B3 DifferentialWheels
DEF Y1 DifferentialWheels
DEF Y2 DifferentialWheels
DEF Y3 DifferentialWheels
Supervisor
```

```c
/*
 * File:         soccer_player.c
 * Date:         February 11th, 2003
 * Description:  A controller for players
 * Author:       Olivier Michel
 * Modifications:Simon Blanchoud - Septemb
 *                - Adapting the code to t
 *               Yvan Bourquin - October 2
 *                - Adapted to new packet-
 *
 * Copyright (c) 2006 Cyberbotics - www.cy
 */

#include <stdlib.h>
#include <time.h>
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/receiver.h>

#define ROBOTS 6
#define TIME_STEP 64

/*
 * The following macros are useful for ret
 * angles value for each robot specified b
 * player id: '1', '2', or '3'
 */
#define robot_get_id(t,p) (3*((p)-'1'+(((t
#define robot_get_x(t,p) packet[robot_get_
#define robot_get_y(t,p) packet[robot_get_
#define robot_get_orientation(t,p) packet[
#define ball_get_x() packet[ROBOTS*3]
#define ball_get_y() packet[ROBOTS*3+1]

int main() {
  WbDeviceTag receiver; /* to receive coor
  char team;            /* can be either '
  char player;          /* can be either '
  const char *name;
  int counter = 0, max = 50;
  int left_speed, right_speed;
  double v, d;
```

Figure 2.12: Webots Integrated Development Environment.

2.6.3 Simulation environment overview

The flexibility of Webots^{TM} allows the creation of any kind of simulation environment. The user can create the simulation environment that suits his needs. One example is *Rat's life*[10], a competition to promote research and stimulate further interest in bio-inspired robotics control. Webots^{TM} simulates the simulation environment where two rat robots compete each other for survival in a maze-like environment.

2.6.4 Visualization and interaction

A simulated world in Webots^{TM} is rendered onto the 3D window (Figure 2.12). By itself the 3D window allows the manipulation of the location and rotation of simulation objects. Through the API is possible to extend the interaction with the simulation provided by the 3D window. The console windows (Figure 2.12) can be used with the API to ask for user input and give feedback to the user.

2.7 Gazebo

Gazebo [17, 18] is an open source multi-robot simulator for outdoor environment and is part of the Player Project [19, 20, 21] a de facto standard in the open source robotics

[10]http://www.ratslife.org/

community [22]. It its capable of generating realistic sensor feedback and reasonable interactions between objects.

2.7.1 Software overview

The simulator is developed in the C++ programming language for the Linux operating system. Gazebo's architecture was designed to provide easy creation of new robots, actuators, sensor and arbitrary objects. The result is a simple API for the addition of these *models* and necessary interfaces for interaction with clients. Underneath the API resides the third-party libraries thad handle both the physics simulation and visualization. The third-party libraries interface with Gazebo through an abstraction layer that prevents models from becoming dependent on specific libraries that may change in the future. This architecture is depicted in Figure 2.13.

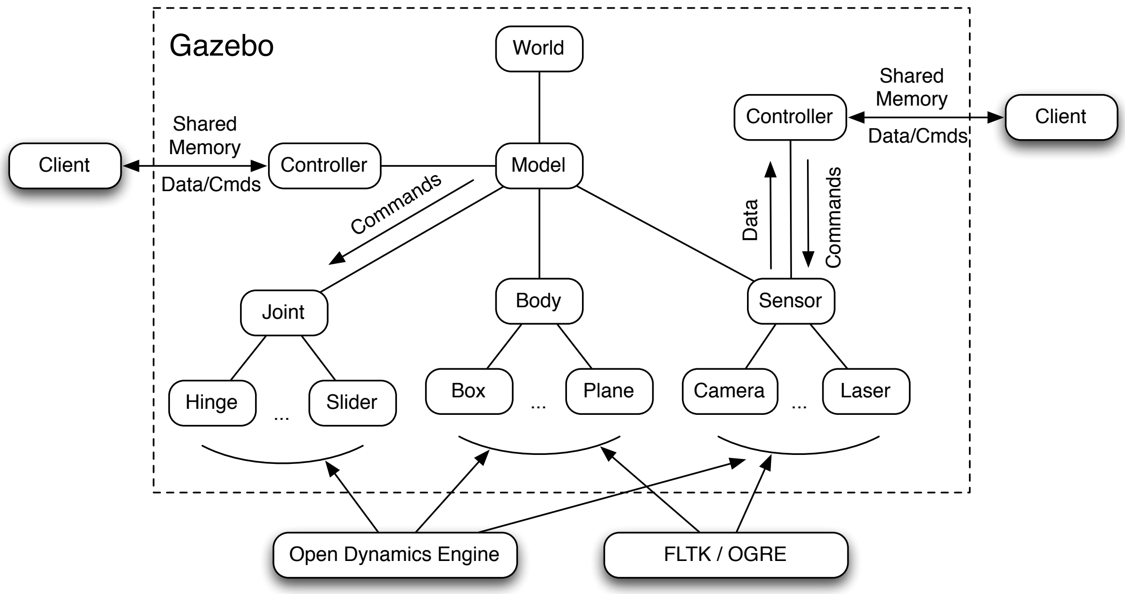


Figure 2.13: General structure of Gazebo components, adapted from [17].

The *World* is the set of all models and environmental factors such as gravity and lightning. Each model has at least one body and any number of joints and sensors. A *model* or sensor can have many actuators (i.e. controllers), each of which has an interface to send data or receive commands, e.g. control joints or transmit the image data of a camera. The client can send commands and receive data through the interface provided by the actuator. The interface is implemented with memory mapped files that create a shared memory region and is compatible with Player [20] clients.

The physics engine in Gazebo is supported by the Open Dynamics Engine (ODE) [6]. It includes several features such as collision detection, joints, mass, several geometries and triangle meshes. By using the abstraction layer between Gazebo and ODE normal and abstract objects can be easily created such as laser rays and ground planes.

The visualization engine has two components a GUI and rendering component. The GUI is provided by Fast light Toolkit (FLTK) [23] a cross-platfom C++ GUI toolkit. For rendering the *World* gazebo uses Object-Oriented Graphics Rendering Engine (OGRE) [24] a scene-oriented, flexible 3D engine written in C++.

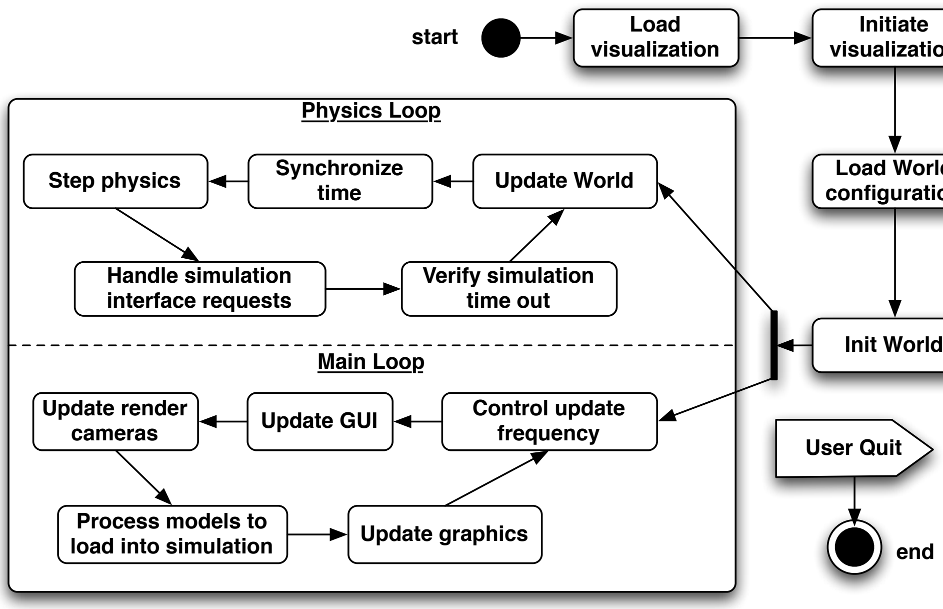


Figure 2.14: Overview of the Gazebo workflow.

An overview of the simulator workflow is shown on Figure 2.14. When the simulator starts it begins by loading the GUI and the rendering engine (i.e. OGRE) and then initiates them, that is, creates them and shows them to the user. The user can opt for loading the rendering without the GUI but the inverse is not possible, also the visualization can be disabled because it is not required for the simulation to run. Afterwards, from the same configuration file, it loads the simulation *World* that defines the *models* and configures the physics engine and then initiates them.

The main loop and the physics loop run concurrently. The physics loop is responsible for updating the world state and the physics simulation. The time synchronization provides control over the simulation time, it can be realtime or faster/slower than realtime. The user may define a simulation time terminus, therefore to oblige with the request the physics loop will issue a *user quit* signal after *timeout* simulation seconds, a *timeout* equal to zero (the default value) means no timeout. The main loop, at each cycle, is responsible for updating the visualization and triggering the loading of new models into the simulation during runtime. The frequency at which the cycle updates is hardcoded and controlled within the cycle.

2.7.2 World representation

In Gazebo the *World* is a set of *models* defined in an Extensible Markup Language (XML) file. A *model* is any object with a persistent physical representation. This allows the build of anything from simple geometries to complex robots. *Models* have at least one rigid body, zero or more joints and many sensors and controllers.

Bodies are the building blocks of a *model.* The arrangement of geometric shapes chosen from boxes, cylinders, planes, spheres, rays, maps, height-maps and triangle meshes models the physical representation the bodies. Each geometry has several physical attributes such as mass, friction, bounce factor and rendering properties such as color, texture, mesh, etc. On a collision, because both geometries carries the parameters that define how a collision is handle, the simulator will chose the lowest values to be used in the collision contacts (Section 2.2.4). By associating a mesh to a body we can define a *skin* that will be rendered to represent the body instead of the geometries, an example is shown in Figure 2.15.

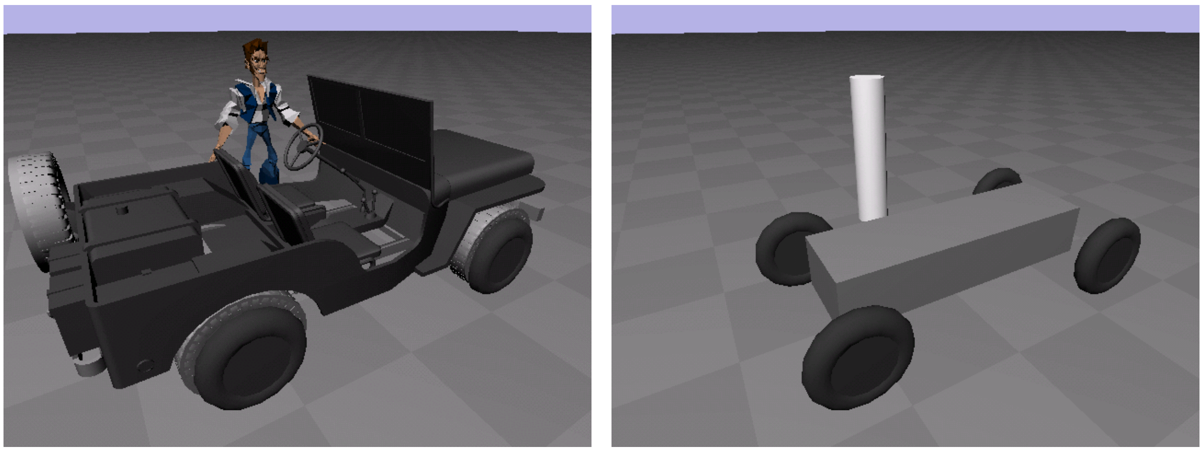

(a) Models rendered with a skin (b) Models rendered without a skin

Figure 2.15: Model rendering in Gazebo, from [18].

Joints are the glue that connect bodies together to create kinetic model and dynamic relationships. Several joint types are available such as hinge joints that provide rotation along one or two axis, ball and socket joints, etc. Joints can also act as motors, the friction between a connected object and other causes motion.

Sensors are abstract devices without physical representation but become tangible when incorporated in a model. This allows the reuse of the same sensor in multiple models. Gazebo has several sensors implementations including a mono camera, stereo camera, contact sensor, inertial measurement unit sensor, infra-red sensor and ray sensor.

Actuators provide the mechanism by which a client can access and control a model. An actuator can access several components of a model, e.g. joints and sensor, and recreate a certain device. For example, an actuator can embody a robotic arm by controlling several joints with the appropriate constraints and kinematic model. Gazebo includes actuators (i.e controllers) such as bumper, differential steering, holonomic motion, gripper, etc.

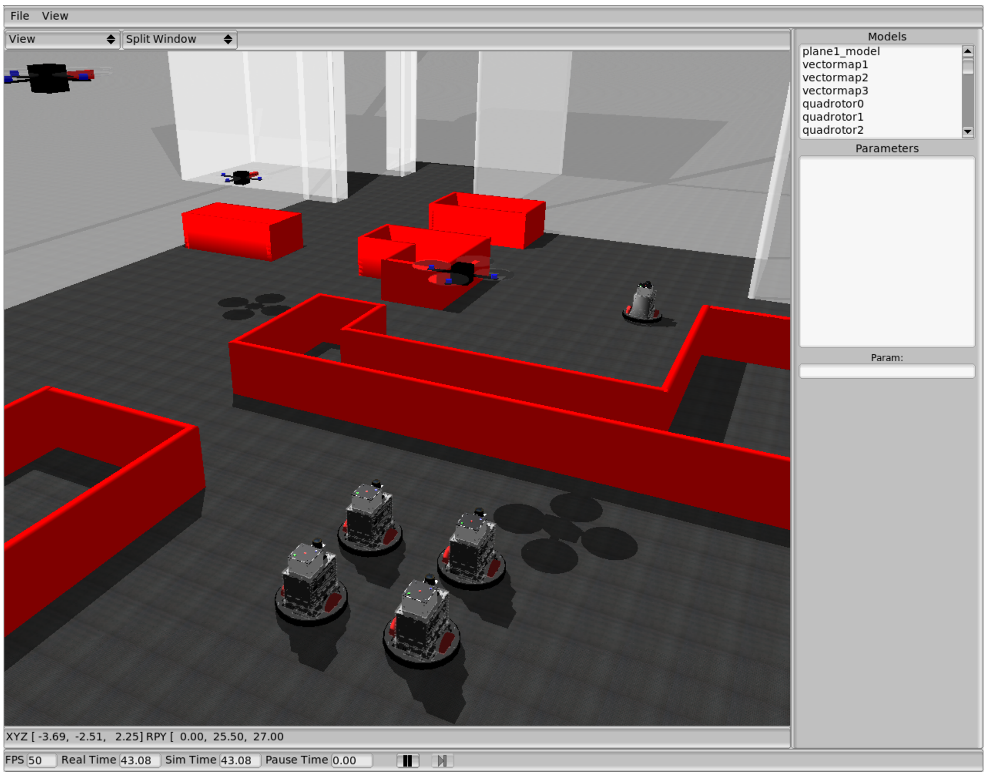
2.7.3 Simulation environment overview

The flexible *models* of Gazebo allows the creation of any kind of simulation environment. The user can create the simulation environment that suits his needs. It can be an outdoor environment with a rough terrain for a search and rescue robot or a simulation environment for the robots of the RoboCup MSL.

All *models* are bound to the physics engine that simulates the three-dimensional dynamics of objects. During the simulation the *models* can be controlled by clients that connect to the interfaces provided by them. The sensors and actuators have update cycles with a frequency defined by the user. The sensors generates data by querying the world state.

2.7.4 Visualization and interaction

As mentioned, the visualization of Gazebo has two components, the GUI and the rendering engine (Figure 2.16). Both components work together to provide the user a visualization of the simulation, at the same time, they show the information of the selected *model* in the rendering widget and the status of the simulation. From the GUI it is possible to stop and resume the simulation and modify the state parameters value of the the selected *model*.



```
plane1_model
vectormap1
vectormap2
vectormap3
quadrotor0
quadrotor1
quadrotor2
```

```
XYZ [ -3.69, -2.51,  2.25] RPY [  0.00,  25.50,  27.00
FPS 50   Real Time 43.08   Sim Time 43.08   Pause Time 0.00
```

Figure 2.16: Gazebo's visualization window, from [25].

The simulator provide several interfaces for clients to access and control the simulation. The interfaces have the same technical implementation of the *models* interfaces. The user can access and modify the simulation and *models* state, including starting and resuming the simulation, repositioning *models*, getting *models* three-dimension pose, etc.

2.8 SimSpark

SimSpark [26] is a multi-robot simulator based on Spark [27] a generic physical multi-agents simulator. It is used as the official RoboCup 3D simulation server [28].

2.8.1 Software overview

The SimSpark simulator is developed in the C++ programming language with implementations for Windows, Linux and Mac OS X operating systems. It is built around three main components, they are the physics engine, the object and memory management system (*Zeitgeist*), and the simulation engine. The overall architecture is shown in Figure 2.17.

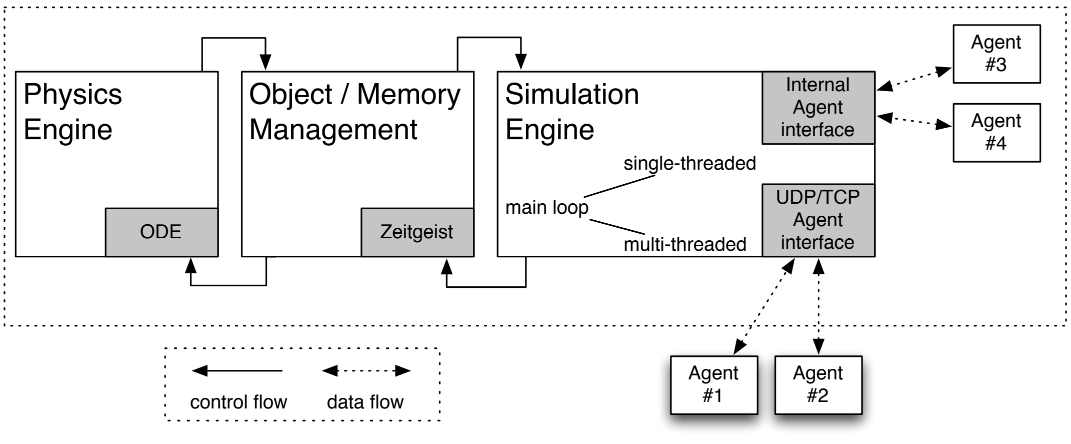
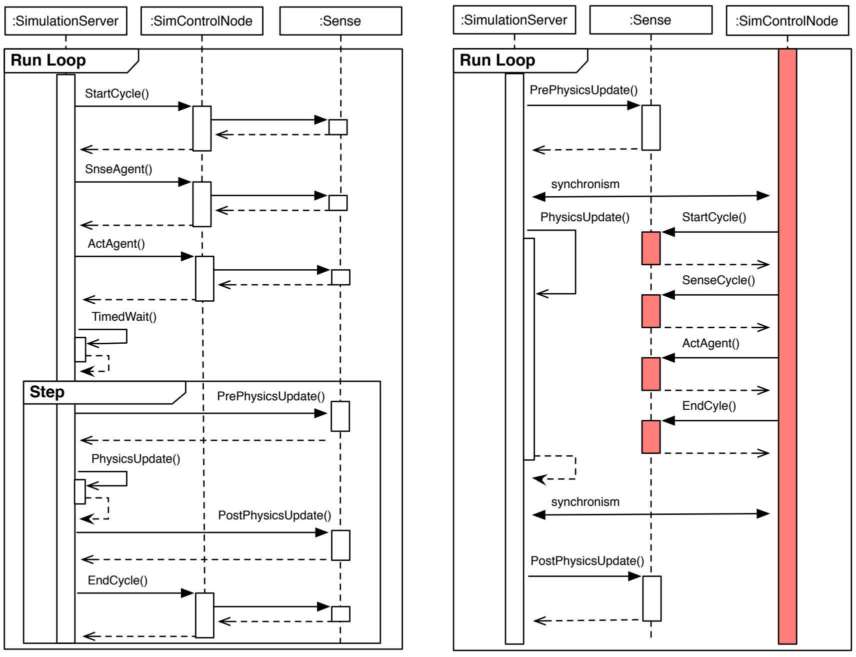


Figure 2.17: Overall architecture of SimSpark. Control flow and data flow between the main components of the simulator, adapted from [27]. The graphics component is not shown because it is an optional part of the system.

The current implementation of the physics engine uses the Open Dynamics Engine (ODE) for dynamics simulation. It provides rigid body simulations, collision detection and joints for articulated body structures. To maintain the object-oriented design of the simulator the ODE functionalities are encapsulated in classes.

Zeitgeist is an application framework for handling data objects and functional parts of a system in an uniform way. Its implementation is based on two concepts. The first concept follows a variant of the reflective factory pattern [29] where a factory instantiates object at runtime while storing information about the factory itself in the object. This can be used to determine the class name and supported interfaces of a recently created objected.

This is the support for the scripting language of the simulator. Currently Ruby[11] is the only supported scripting language. The second concept is the organization of the factories objects and created objects in a virtual file system. By providing a path expression services and objects can be easily located at runtime. The object factories are located in well-defined locations which allows the instantiation of object of unknown classes at compile time, e.g through the scripting language interface. This allow the addition of new sensors, actuators and other things to the simulator in the form of plugins. More information about the *Zeitgeist* framework can be found in [27].



(a) Single-threaded loop. (b) Multi-threaded loop.

Figure 2.18: SimSpark simulation loop, from [28].

The simulation engine handles timing, event management and communications with external entities. It was designed to allow a customization of the runloop with replaceable components. There were two built-in runloops: a simple loop that would execute actions sent by the agents as soon they arrived, and a more complex one using SPADES [30], designed to support distributed simulations. The SPADES runloop turned out to be too complex to allow the system to mature enough to be use in competition. The simple loop was too indeterministic for accurate control and suffered large time variations based on the load of the server. Therefore, a new runloop mode was implemented with a control loop that runs at 50 Hz, the rendering at 25 Hz, and different delays for different sensors and actuators. Furthermore, a single-threaded mode and multi-threaded mode of the run-loop was implemented (Figure 2.18). In the multi-threaded mode the physical simulation

[11]`http://www.ruby-lang.org`

stepping and the agent communications processing for the next simulation cycle run in parallel. The game or application logic of the simulation can be implemented as plugins triggered by certain event types to which they can chose to respond.

Agents and external monitors for visualization can connect to the simulator via TCP and UDP connections, although only TCP is used at the moment. There is also an interface for a *trainer* application that has special permissions to move objects in the simulation, set game states, etc. This is useful for debugging and machine learning applications.

2.8.2 World representation

In SimSpark, besides the internal C++ interface, the *World* can be defined by using the Ruby interface or by a scene description language called *RubySceneGraph* (RSG). The language is based on S-Expressions [31] and its hierarchical parenthesis structure is mapped into an object hierarchy of the scene tree. The Ruby interface and RSG descriptions can be used together to construct scenes procedurally and descriptively at the same time. For a detailed specification, please refer to [27].

Entities in a scene graph are represented with a node containing the reference coordinate and orientation of the entity. An entity has different properties like physics or geometries, and are called *aspects* and are represented as nodes in the scene graph. Physics *aspects* constitute physical properties of a body, mass and mass distribution. Geometry *aspects* constitute the colliders that implement the shape of objects to handle collisions with other objects. Further *aspects* describes how a node have to be rendered on screen. It is possible to load triangle meshes to provide a more detailed visualization of a node.

Joints, represented by special nodes, can be used to constrain the relative movement of two connected bodies along one or more axis. Also, joints acting as motors can be used to enforce movement. The connected bodies form an articulated structure that can be used to simulate wheeled or legged robots. A joint can be created and stored in the scene graph, several joint types are available including hinge joint, two-hinge joint, ball and socket joints, slider joint and universal joint.

An external agent is represented inside the simulator with an agent proxy. It collects sensor data and executes actions on behalf of a connected remote agent. An agent *aspect* identifies a subtree of the scene as an agent. The physical and geometry *aspects* are no different from other objects in the simulation. An agent can have perceptors and effectors, represented by nodes in the scene graph. The perceptors provide the sensors data to the agent associated with the representation of the agent in the simulator, and the effectors can be used by the agent to act in its environment.

2.8.3 Simulation environment overview

The flexibility of the SimSpark architecture allows the creation of any kind of simulation environment. One example is the the humanoid soccer simulator with the Aldebaran Nao robot [32] model. The simulator, called *rcssserver3D*, is based on SimSpark and creates the adequate simulation environment for humanoid soccer. It creates a game field, a ball, the goals, defines the Nao robot model and has a plugin that automatically enforces the game rules. Each time an external agent connects to the simulator a new Nao robot model is added to the simulation that serves as representation of the agent that just connected.

2.8.4 Visualization and Interaction

SimSpark has a 3D visualization application, called *monitor* responsible for rendering the simulation scene (Figure 2.19). The *monitor* can be internal or external. The internal *monitor* is part of the SimSpark server while the external *monitor* is a separated application. The external *monitor* connects to a running SimSpark server or replays a simulation from a log file. The data stream sent from server or in the log file has a format called *Monitor Format* [33], a language used to described the simulation state. The binding of an action, executed by the monitor, to a key or mouse event can be customized in the configuration files. Besides the common manipulation of the rendering point-of-view, the monitor can send commands to the simulator that affects the state of the simulation objects.

Additional interaction with the simulation is achieved with Ruby interface that allows the creation of plugins with access to the simulation objects.

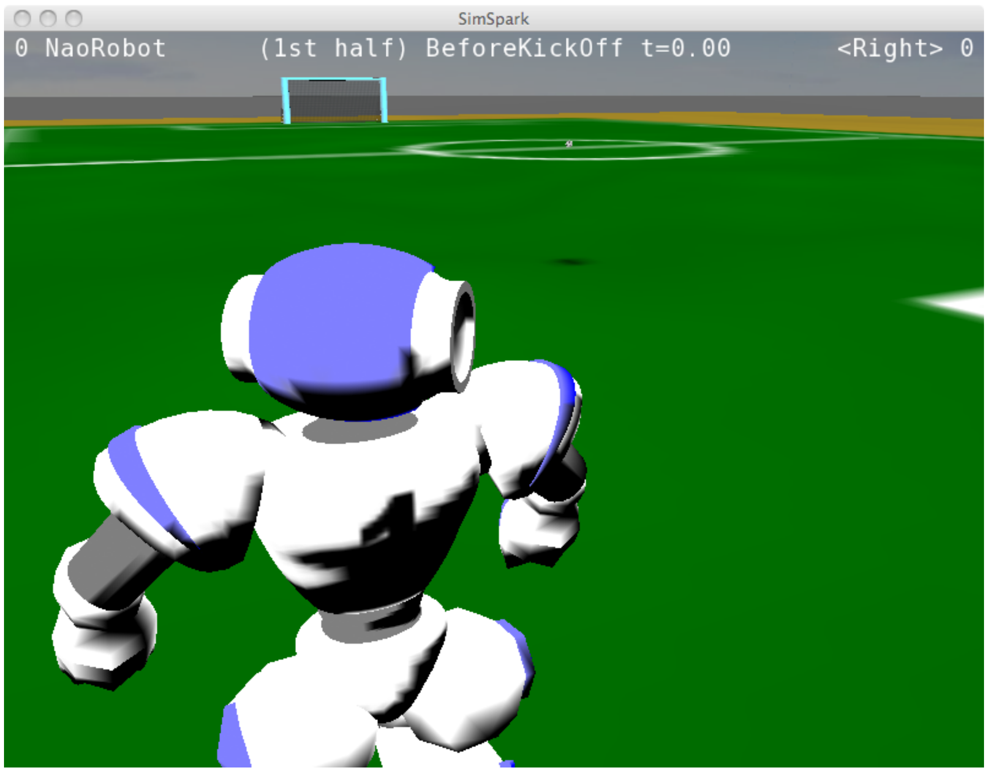


Figure 2.19: SimSpark monitor.

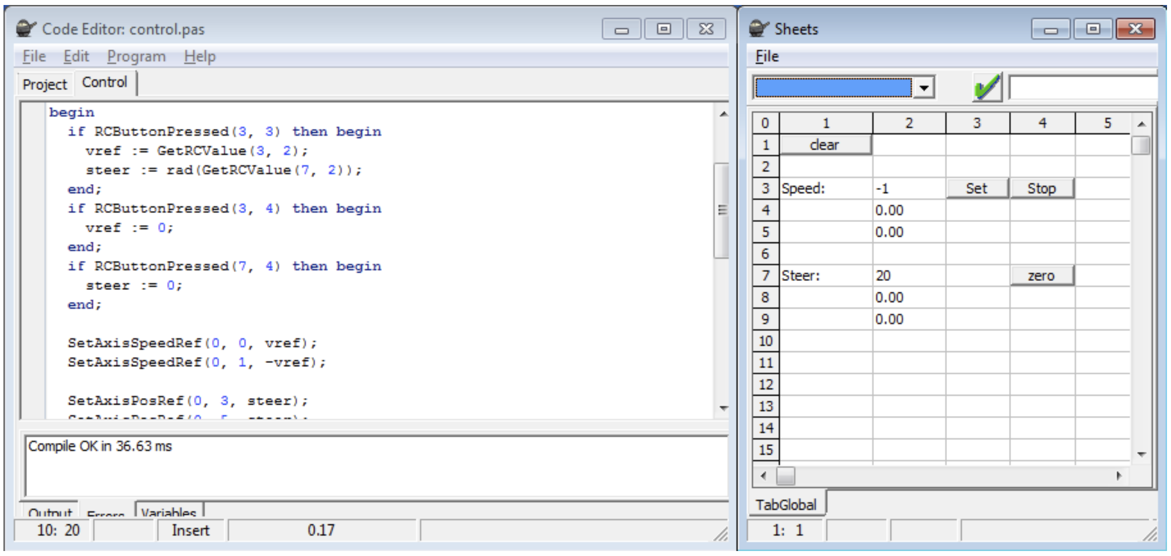
2.9 SimTwo

SimTwo [34] is a realistic simulation system for implementing several types robots, including wheeled, legged and flying robots.

2.9.1 Software overview

The SimTwo simulator is a free to use application for the Windows platform. It has a three-dimensional rigid body dynamics provided by the library ODE. The realistic dynamics is achieved by decomposing the robot in a system of rigid bodies and electrical motors. For visualization purposes it uses GLScene[12] to draw the three-dimensional environment. The robots can be controlled remotely with a communications mechanism via UDP or serial port.

The simulator also offers an interface to control the robots within the simulation, using *RemObject Pascal Script* [13] it provides a scripting system that can be used to access and control several aspects of the simulated robots, such as position, motors state, etc. The scripting system includes a spreadsheet like mechanism that allows the user to raise events, set input parameters and print outputted data from the script application. To assist the development of control scripts, SimTwo has a built-in script editor with more editing capabilities than a regular editor. The built-in script editor and the sheets GUI are shown in Figure 2.20



```
begin
  if RCButtonPressed(3, 3) then begin
    vref := GetRCValue(3, 2);
    steer := rad(GetRCValue(7, 2));
  end;
  if RCButtonPressed(3, 4) then begin
    vref := 0;
  end;
  if RCButtonPressed(7, 4) then begin
    steer := 0;
  end;

  SetAxisSpeedRef(0, 0, vref);
  SetAxisSpeedRef(0, 1, -vref);

  SetAxisPosRef(0, 3, steer);
```

```
Compile OK in 36.63 ms
```

0	1	2	3	4	5
1	clear				
2					
3	Speed:	-1	Set	Stop	
4		0.00			
5		0.00			
6					
7	Steer:	20		zero	
8		0.00			
9		0.00			
10					
11					
12					
13					
14					
15					

Figure 2.20: SimTwo built-in script editor and sheets GUI.

To help debugging a simulation run, the simulator can generate in real-time a graph chart with the internal state values of the many components of a robot, shown in Fig-

[12]http://glscene.sourceforge.net
[13]http://www.remobjects.com/ps.aspx

ure 2.21. The user can chose to plot only the states that he or she is interested in analyzing. The data used to generate the graph chart can be dumped into a log file.

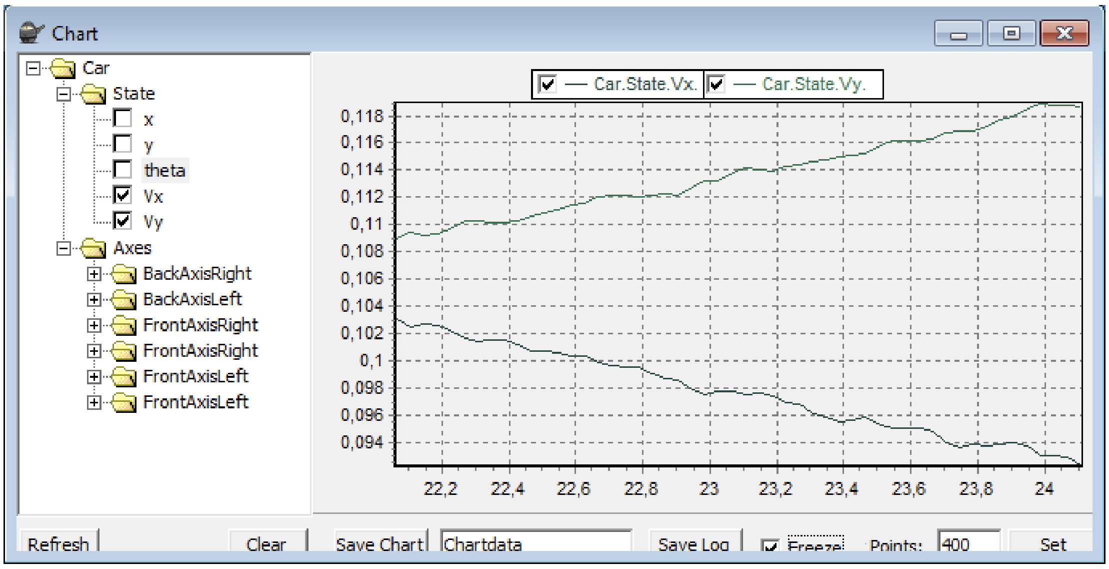


Figure 2.21: SimTwo chart generation.

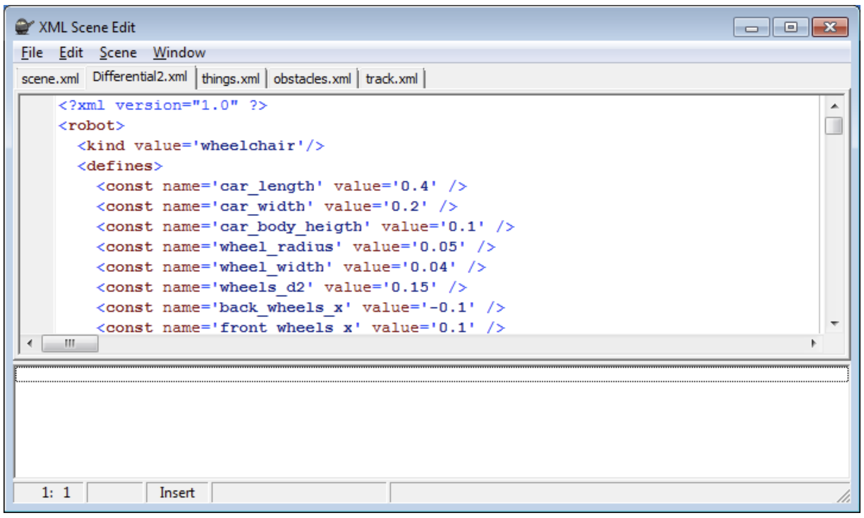
2.9.2 World representation

The simulation world, called scene, is defined by a XML file loaded when the simulation is launched. In the XML file, robots, obstacle, objects and tracks can be defined. The scene files can be created and modified in a built-in scene editor, shown in Figure 2.22

The robot is built by a series of elements that will define its structure and dynamic. The physical structure of the robot takes the form of geometric solids used for dynamics and collision detection, such as spheres, cylinders and cuboids. Articulated components are created by defining joints that create constraints between two geometric solids. A joint can be controlled by a motor component that represent a motor, a gear box and low-level PID controller. Sensors can also be attached to the robot. To give more complex shapes to the robot, a set of plates can be define to create a shell for the robot. Also, a shell helps overcome one limitation of ODE, which is the lack of support for collisions between cylinders. A set of wheels can can also be added to the robot, each of which has a motor, a controller, an encoder and a cylinder that can represent a omnidirectional wheel or not.

Obstacles and objects are represented by geometric solids. The difference between an obstacle and an object is that the first is static and the second is free to move in the environment.

Tracks can be used to draw markers in the floor, e.g a racing track. Several draw primitive are available, including lines, arcs and polygons.



```xml
<?xml version="1.0" ?>
<robot>
  <kind value='wheelchair'/>
  <defines>
    <const name='car_length' value='0.4' />
    <const name='car_width' value='0.2' />
    <const name='car_body_heigth' value='0.1' />
    <const name='wheel_radius' value='0.05' />
    <const name='wheel_width' value='0.04' />
    <const name='wheels_d2' value='0.15' />
    <const name='back_wheels_x' value='-0.1' />
    <const name='front wheels x' value='0.1' />
```

Figure 2.22: SimTwo built-in scene editor.

2.9.3 Simulation environment overview

The flexibility of SimTwo definition allows the creation of different robotic simulation environments. The SimTwo simulator package includes several examples of simulation environments, such as autonomous driving and humanoid simulation.

All geometric solids are bound to the physics engine that simulates the three dimensional dynamics of objects. The control model is made in two levels. In the first level, the control of the motors is made internally by the simulator at every 10ms. At a lower frequency (usually 40ms), more complex controllers are implemented by calling a function found in a script, or by exchanging UDP packets.

2.9.4 Visualization and interaction

The simulation environment in SimTwo is rendered onto the 3D windows (Figure 2.23). From the 3D window the user can manipulate the rendering point-of-view and the location of objects (e.g robots, ball). Furthermore, a configuration window provides additional control over the simulation and allows run-time configuration of the graphics engine, physics engine and serial port communication.

From the control pane (Figure 2.24a), the user can freeze the simulation, change the robots location and edit joints waypoints. From the graphics pane, the user can configure the level of detail of the rendered scene, and control the camera position and orientation.

2.10 Other robotic simulators

Many robotics simulators are available and can be used to developed a simulated environment for a robotic soccer agent. However, because it is not viable to describe all

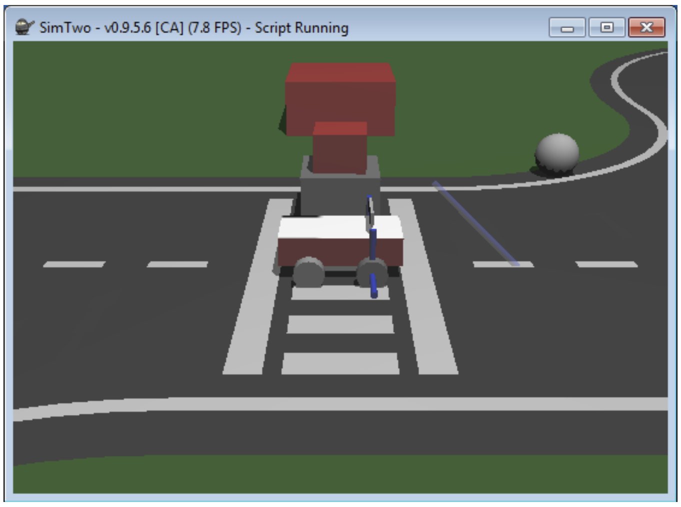


Figure 2.23: SimTwo rendering window.

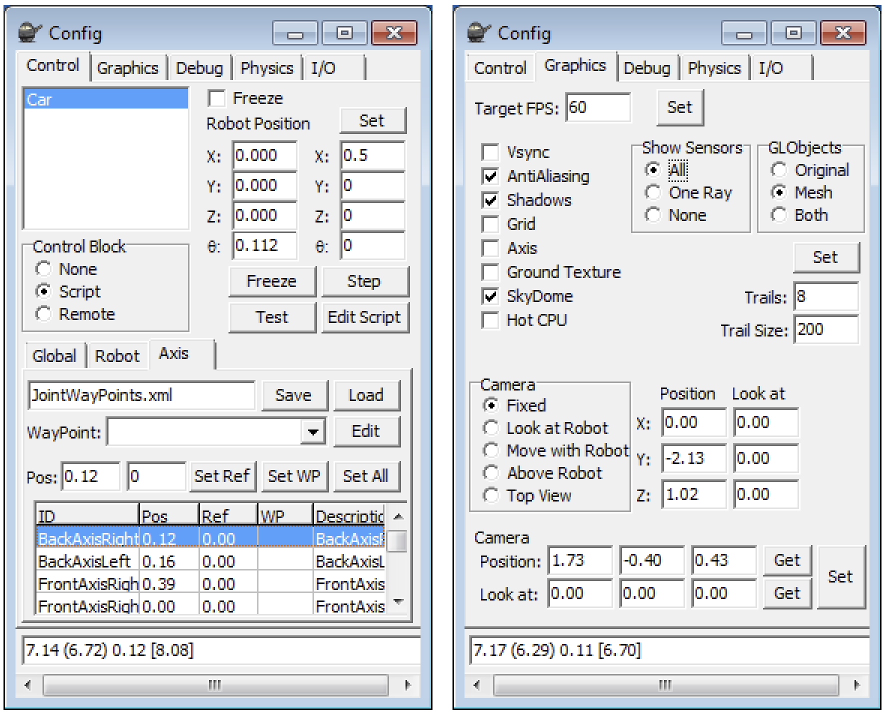
existing simulator, in this section we will briefly describe other robotics simulators that complies with the selection criteria defined in Section 2.1:

SimRobot is a multi-platform robotic simulator which is able to simulate arbitrary user-defined robots in three-dimensional space [35]. It includes a physical model supported by the Open Dynamics Engine (ODE), a graphics engine, a GUI, sensors and actuators. The specification of the robots and the environment, called *scene*, is modeled via an external XML file loaded at runtime. SimRobot has been used to simulate the robots and the environment of the Sony Four-legged League and SPC league.

Microsoft Robotics Developer Studio (MRDS)[14] is a Windows-based environment for academic, hobbyist and commercial developers to easily create robotics applications across a variety of hardware. Based on *Concurrency and Coordination Runtime*, a library to manage asynchronous parallel tasks, it provides a visual programming tool for creating robot applications, web-based interfaces, 3D simulations and a set of sensors and actuators.

ÜberSim is a vision-centric robot simulator [36] used in the RoboCup domain. The simulator core was writen in C++ fo the Windows platform. It includes a realistic robot dynamics and simulation accuracy supported by the Open Dynamics Engine (ODE), a graphics engine, sensors and actuators. It maintains the traditional client/server paradigm where communication is made thought TCP sockets. It provides flexible and extensible robot specification. Vision synthesis is handled by the combination of OpenGL and Open Scene Graph (OSG)[15] library.

[15]http://www.openscenegraph.org



Car

[ ] Freeze

Robot Position

X:	0.000	X:	0.5
Y:	0.000	Y:	0
Z:	0.000	Z:	0
θ:	0.112	θ:	0

Control Block: ○ None [✓ Script] ○ Remote

JointWayPoints.xml

WayPoint:

Pos: 0.12 0

ID	Pos	Ref	WP	Descriptic
BackAxisRight	0.12	0.00		BackAxisf
BackAxisLeft	0.16	0.00		BackAxisL
FrontAxisRigh	0.39	0.00		FrontAxis
FrontAxisRigh	0.00	0.00		FrontAxis

7.14 (6.72) 0.12 [8.08]

Target FPS: 60

[ ] Vsync
[✓] AntiAliasing
[✓] Shadows
[ ] Grid
[ ] Axis
[ ] Ground Texture
[✓] SkyDome
[ ] Hot CPU

Show Sensors: [✓ All] ○ One Ray ○ None

GLObjects: ○ Original [✓ Mesh] ○ Both

Trails: 8

Trail Size: 200

Camera: [✓ Fixed] ○ Look at Robot ○ Move with Robot ○ Above Robot ○ Top View

	Position	Look at
X:	0.00	0.00
Y:	-2.13	0.00
Z:	1.02	0.00

Camera

Position: 1.73 -0.40 0.43

Look at: 0.00 0.00 0.00

7.17 (6.29) 0.11 [6.70]

(a) Control pane (b) Graphics pane

Figure 2.24: SimTwo configuration window.

2.11 Comparison between robotic simulators

The aforementioned robotic simulators are different but with common features. Having their features compared gives us a quick reference of their capabilities and help us chose the best tools for the job in hands. In the context of this thesis we defined a set of requirements (Section 1.3.1 and 1.3.2) for our robotic simulator. Therefore, it is only appropriated to use these requirements as our frame of reference for comparison. Although, we excluded the CAMBADA agent integration requirement because it is foreseeable that only a robotic simulator created or adapted for the CAMBADA team will comply with this requirement. The comparison is presented in Table 2.1.

2.12 Summary

After reading this chapter, the reader should understand the purpose of a robotic simulation and its application on the RoboCup domain. Different robotic simulators solutions were presented and some of their characteristics described. In the end we presented a comparison table between the described robotic simulators.

	Rigid Body Dynamics	Multiple entities	Sensors / Controllers	Flexible models	Licence	Operating system	Visualization and Interaction
Brainstormers Tribots	Yes (3D)	Yes but limited	No	No (hardcoded)	Open Source	UNIX-like	Yes
Robocup Simulator Dev	Yes (3D)	Yes but limited	No	No (hardcoded)	Open Source	Windows	Yes
SimSrv	Yes (2D)	Yes	Yes	Yes	Open Source	UNIX-like	Yes
Webots	Yes (3D)	Yes	Yes	Yes (VRML)	Commercial	Win/Linux/Mac	Yes
Gazebo	Yes (3D)	Yes	Yes	Yes (XML)	Open Source	Unix-like	Yes
SimSpark	Yes (3D)	Yes	Yes	Yes (RSG/Ruby)	Open Source	Win/Linux/Mac	Yes
CAMBADA simulator	Yes (3D)	Yes but limited	No	No (hardcoded)	Open Source	UNIX-like	Yes
SimRobot	Yes (3D)	Yes	Yes	Yes (XML)	Open Source	Win/Linux/Mac	Yes
MRDS	Yes (3D)	Yes	Yes	Yes	Freeware	Windows	Yes
Ubersim	Yes (3D)	Yes	Yes	Yes (XML)	Closed Source	Windows	Yes
SimTwo	Yes (3D)	Yes	Yes	Yes (XML)	Freeware	Windows	Yes

Table 2.1: Comparison between robotic simulators.

Chapter 3

CAMBADA

The Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture (CAMBADA) [37] is the RoboCup MSL soccer team of the University of Aveiro, Portugal. This project started officially in October 2003. The team is composed by six robots and the first version was completed in 2004 and since then there has been a steady evolution in their structure to face new challenges and higher goals. This chapter gives an overview of the RoboCup competition, giving a special attention to the MSL, and describes the current stage of the CAMBADA robots.

3.1 RoboCup

RoboCup [38] is an international project created to promote research and education in the field of Artificial Inteligence (AI), robotics and related areas. Soccer was chosen as a real world challenge where a wide range of technologies can be integrated and examined. The long term goal is to, circa 2050, develop a fully autonomous humanoid team capable of playing and wining against the human world championship soccer. To fulfill this goal RoboCup is divided in various competitions which are divided in leagues:

RoboCup Junior

"This is a project focused on education and it is designed to introduce robotics to young students still in primary and secondary school, as well undergraduates who do not have the resources to get involved in the senior leagues" [39]. This competition has three leagues:

- Soccer
- Dance
- Rescue

RoboCup Rescue

"Search and rescue of victims in a disaster scenario is a very important social issue which involves a very large number of heterogeneous agents in the hostile environment. The purpose of this project is to promote research and development in this significant domain" [40]. This project has two leagues concurrently proceeding:

- Robot League
- Simulation League
  - Agent Competition
  - Infrastructure Competition
  - Virtual Robot Competition

RoboCup Soccer

"The soccer games are important opportunities for researcher to test their work and exchange technical information. It also serves a good opportunity to educate and entertain the public". This competition is divided into the following leagues:

- Humanoid League
  - Kid Size
  - Teen Size
  - Adult Size
- Middle Size League
- Simulation Leagues
  - 2D Simulation
  - 3D Simulation
  - 3D Development
- Mixed Reality
- Small Size League
- Standard Platform League

RoboCup @Home

"This competition aims to develop service and assistive technology for future personal domestic applications" [41].

Although soccer is the main application domain, each competition has its own singularity that enables researches to address different problems in different areas of robotics such as multi-agents system coordination, autonomous agents, real-time reasoning, strategy coordination, reactive behavior, real-time sensor fusion, machine learning, computer vision, motor control and intelligent robot control.

3.1.1 Middle Size League (MSL)

CAMBADA robots were design to compete in RoboCup's MSL. In this league each team can play with up to 5 robots with maximum dimension of $50cm \times 50cm \times 80cm$ and a maximum weight of 40Kg to play in a field of $12m \times 18m$ depicted in Figure 3.1 according to robot-adapted official FIFA rules. The robots must have their sensors on-board and be able to operate fully autonomously. To have cooperative behaviors, robots are allowed to communicate with team-mates and the coach through wireless connection. The coach must be an autonomous entity, usually an external computer, without sensors and with the capability of deciding accordingly the informations provided by its team-mates. The referee is the only human decider which enforces the game rules. The robots plays in an environment well defined by colors, such as green for field, white for line and mainly black for robots. Since 2010 edition, the ball is no longer of orange color, it can be a generic FIFA approved ball.

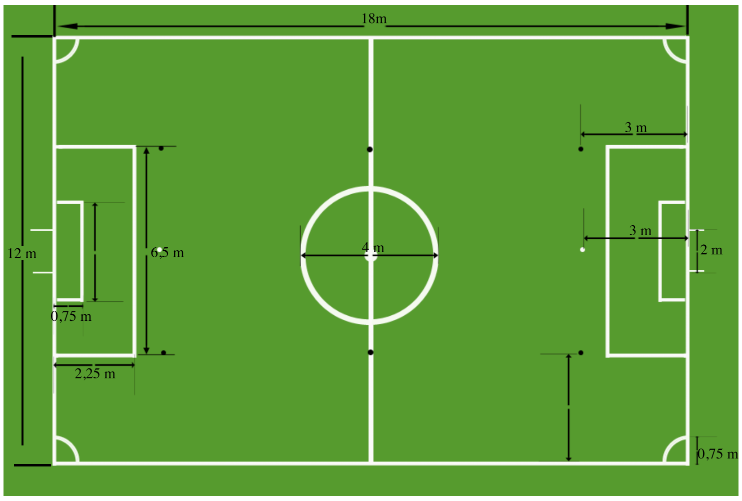


Figure 3.1: A Middle Size League field with official markings.

3.2 General Architecture

The general architecture of the CAMBADA robots is described in [42, 43] Basically, the CAMBADA robots follow a biomorphic (Figure 3.2) paradigm centered on a processing unit (i.e. *the brain*) responsible for the high-level coordination layer. The processing unit handles communication with other robots and also has the vision, a high bandwidth sensor, directly connected to it. The sensing information of low bandwidth and actuating commands to control the robot are sent and received respectively by means of a low-level sensing/actuating system.

The coordination layer works around the RTDB, it contains the local state of the robot as well a remote copy of a subset of the other robots states [42]. The process that handles

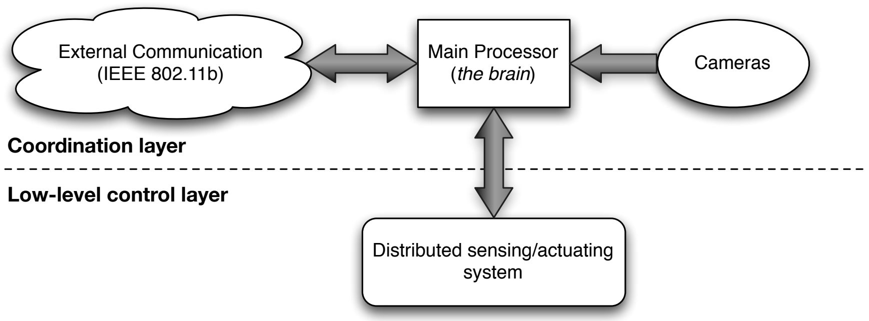


Figure 3.2: The biomorphic architecture of the CAMBADA agents, adapted from [42].

communications updates the remote information of the other robots via an IEEE 802.11b wireless link. Then, at each instance, another set of process uses the RTDB to define a specific robot behavior generating commands that are sent to the low-level control layer through a gateway (Figure 3.3).

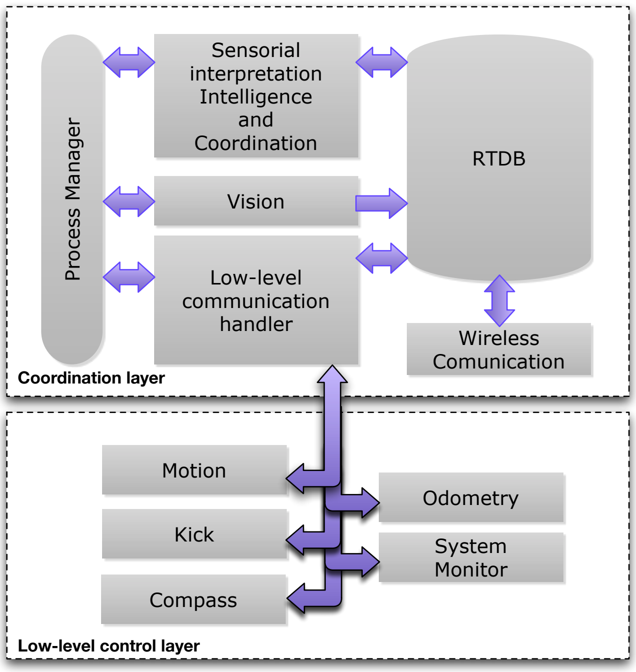
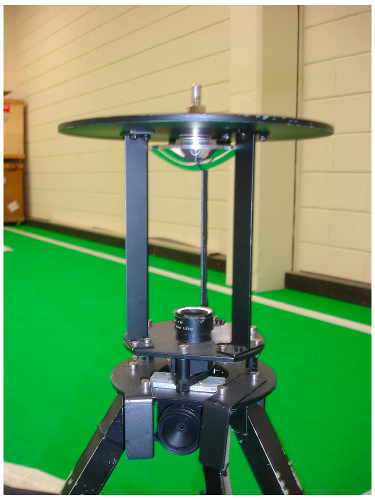
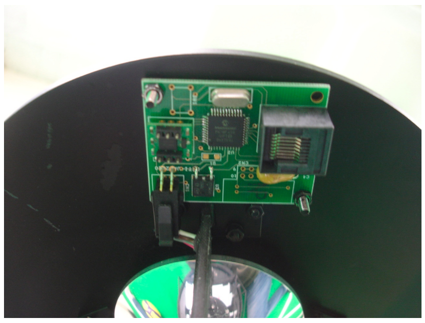


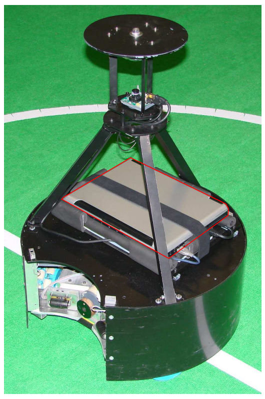
Figure 3.3: Layered software architecture of CAMBADA players, adapted from [37].

3.3 CAMBADA Hardware

The physical structure of the robot is built on a modular approach [44] with three main layers [45]. The top layer holds the vision system of the robot and also an electronic compass, shown in Figure 3.4. The middle layer is the placeholder for the processing unit, currently a 12" laptop. The vision software along with all high level and decision software executes in the laptop where the sensors data is collected and the commands provided to the actuators are computed. Finally, the lowest layer is composed by the robot motion system and the kicking device. Middle and lower layers are shown in Figure 3.4.


(a) The vision system


(b) The electronic compass


(c) The processing unit

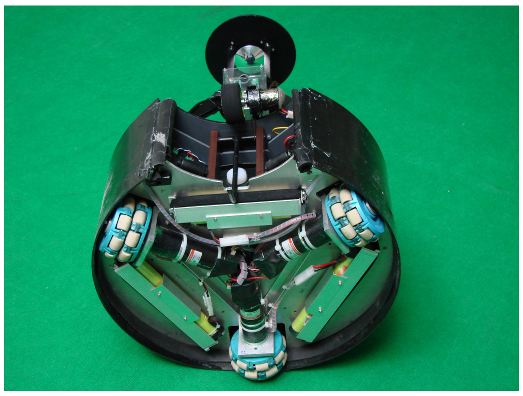

(d) The motion system and the kicking device

Figure 3.4: CAMBADA's physical structure divided in layers. Top layer composed by 3.4a and 3.4b. Middle layer composed by 3.4c. Lower layer composed by 3.4d.

The low level sensing/actuating system control (Figure 3.5), or the robot *nervous system*, is a network of micro-controllers placed beneath the middle layer. Controller Area Network (CAN), a real-time fieldbus typical of distributed systems, was chosen for networking and it uses the FTT-CAN protocol to comply with real-time constraints [46]. Due to the highly distributed nature of the sensing and actuating system, each node in the network controls a different function of the robot [43].

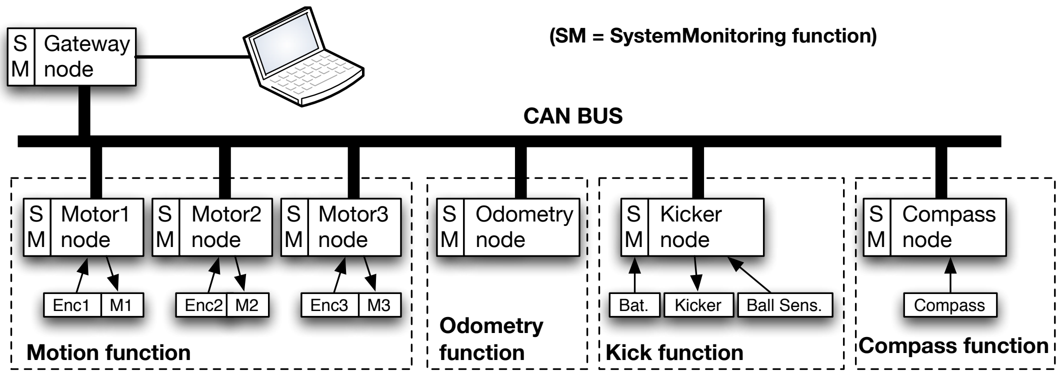


Figure 3.5: Hardware architecture with functional mapping, adapted from [37].

The low level sensing/actuating system executes five main functions, namely Motion, Odometry, Kicker, Compass and System Monitor. The first provides holonomic motion using three DC motors each with a swedish wheel. The Odometry function reads the encoders of the three motors and calculates the robot displacement, afterwards it transmits the information to the coordination layer. The Kick function controls the kicking system and the ball handler system. The Compass function transmits the electronic compass to the coordination layer. Finally, the System Monitor function monitors the robot batteries as well the state of the low-level nodes.

3.3.1 Physical shape

Because the CAMBADA robots compete in the RoboCup MSL, they must not exceed the dimension defined by the rules (see 3.1.1). The shape of the robot is intrinsic to its layers. The top layer has a conical look with a base of $24cm$ of radius and $56,5cm$ of height. The middle layer and the bottom layer build up together a cylindrical shape with $25cm$ radius and $14,5cm$ height with a cut out section to grab the ball, it is also responsible for most of the robot's weight. In a very loose point of view, the physical shape of the robot is a cylinder with a cone on top.

3.3.2 Holonomic Motion

At the bottom of the lower layer three swedish wheels are disposed in a equilateral triangle configuration. This makes all wheels have the same distance to the robot center and have a 120 degrees difference from each other. Assuming that the front of the robot is along the YY axis, the first wheel is positioned at 30 degrees from the XX axis, consequently the second is at 150 degrees and the last at 270 degrees (or -90 degrees). Each wheel is assembled at the shaft of a 24V/150W Maxon motor. The described configuration provides the robot an omnidirectional motion [47]. The Holonomic Motion system is shown in Figure 3.6.

The desired speed for each motor is sent from the coordination layer and applied by the Motion function at the low level sensing/actuating system.

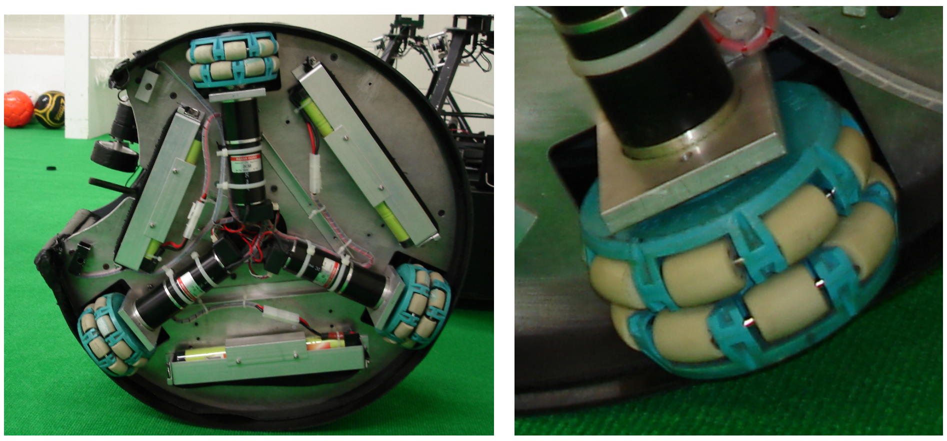

(a) Disposition of the motion system (b) Detailed view of an omni-wheel

Figure 3.6: Holonomic motion system.

3.3.3 Odometry

The odometry resides in the Odometry function at the low level sensing/actuating system. It reads the encoders of the three motors and tries to estimate the robot displacement, the information is then sent to the coordination layer.

3.3.4 Electronic compass

The electronic compass is used to obtain the orientation of the robot. It reads the earth magnetic north which is sent to the coordination layer.

3.3.5 Barrier

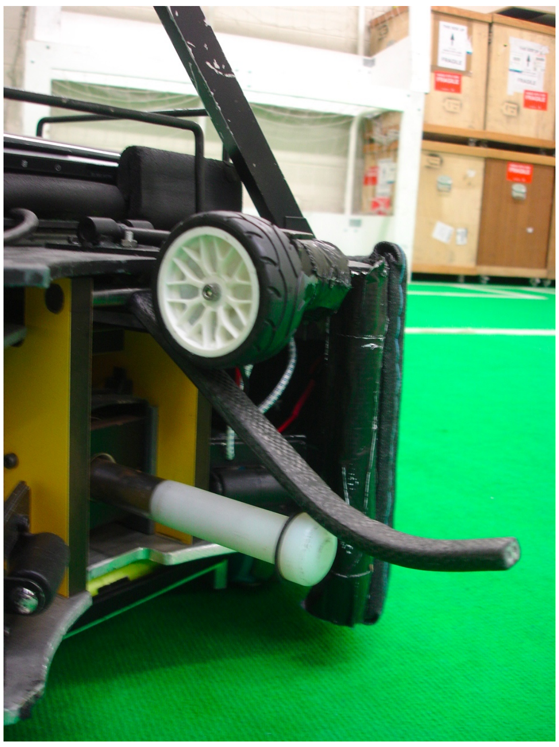
The barrier is a sensor with the purpose of signaling when the ball is under control by the robot. This sensor is integrated in the kicker system.

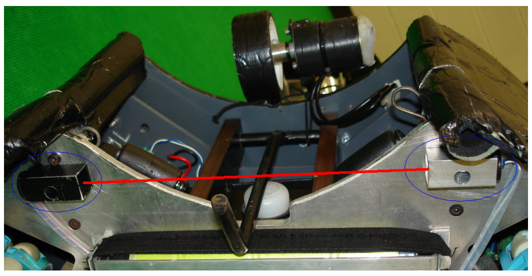
3.3.6 Grabber

The grabber is a simple modeling motor with a wheel design to bring and maintain the ball under control by the robot. When the ball is not under control, which is signaled by the barrier, the wheel is set to spin to bring the ball under control when in range. When the ball is under control the wheel is set to stop. Having the ball under control, then lose the ball and immediately activate the grabber to bring it back again under control an so on, is what we call of maintaining the ball under control. The order to put the wheel spinning is sent by coordination layer and applied by the Kick function. This actuator is integrates in the kicker system.

3.3.7 Kicker

The kicker is an electromagnetic kicking device which allows direct and lob kicking. The granular control of the device's power results in a controllable initial velocity for when the ball is kicked. To avoid kicking the air, the kicker only kicks when the ball is under control which is signaled by the barrier. The power applied to the kicking device is sent by the coordination layer and applied by the Kick function. This actuator plus the barrier and the grabber are the kicker system as a whole, shown in Figure 3.7.


(a) The kicker


(b) The barrier sensor

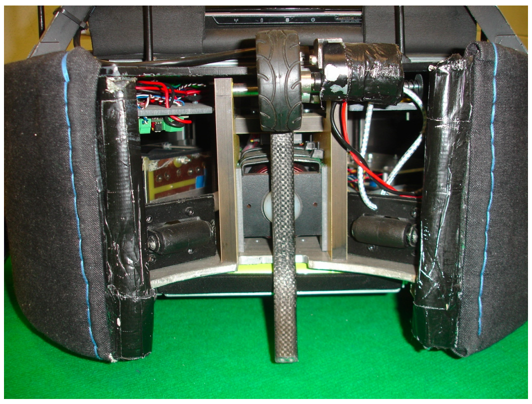

(c) The grabber

Figure 3.7: The kicker system.

3.3.8 Vision system

The catadioptric view of the omnidirectional vision system is obtain by a regular video camera pointed to a hyperbolic mirror. The video camera is a Point Grey camera with 4mm lens attached to the main processor unit, i.e. the laptop, through a firewire interface.

The frontal view is obtain by a regular camera positioned on the robot with the focal axis of the camera parallel to the ground. The frontal video camera is a Point Grey Chameleon camera attached to the main processor unit, i.e. the laptop, through a USB interface, although compatible with the firewire access and manipulation interface. Currently, only the goalkeeper robot has a frontal camera installed.

3.4 CAMBADA Software

Much like the CAMBADA hardware, the software also has a distributed nature (Figure 3.3). The CAMBADA software was – and still is – developed for the GNU/Linux platform, namely Ubuntu, and is deployed at the processing unit, i.e. the laptop also known as *the brain*. Some of the CAMBADA software resides in the coordination layer and is also referred to as high-level software or the coordination layer itself.

The high-level software is the result of a set of processes running concurrently interconnected by shared resources. The shared resources are managed by a software library called RTDB, already mentioned above as the foundation of the coordination layer. The processes synchronization is supported by a library specifically developed for the task, PMan. It supports real-time constraints and precedences among others processes. The constituent processes of the high-level software are "Sensorial interpretation, intelligence and coordination", "Vision", "Low-level communication handler" and "Wireless communication". There is also software that does not make part of the coordination layer but is complementary, including the "'Monitor", the "CambadaConfig" tool and the "Basestation".

3.4.1 Real-Time Data Base (RTDB)

The RTDB provides cooperative sensing by means of a *blackboard* [48], a database where each agent publishes its internal information that maybe accessed by others. The implementation relies on a *distributed shared memory* model and the broadcasting of the local state data, which allows to tackle the problem of fast degradation of data coherence and undesirable delays caused by the typical server-client model [42].

Each database node has two regions, a local and a shared. The local region contains the internal state of the robot, which is used to transmit data between processes. The shared region contains the relevant state of itself and of the other team-mates, the data is broadcasted periodically to the other team members, and consequently updated periodically, to guaranty the coherence (i.e. temporal validity) of the information.

3.4.2 Process Manager layer (PMan)

The CAMBADA software runs on a General Purpose Operating System (GPOS) which lacks some features required by the real-time nature of some applications. The PMan [49] is a user-space library developed to extend the native services already provided by the underlining GPOS, i.e. Linux Kernel[1] in this case. The library provide automatic process triggering, run-time adaptive Quality of Service (QoS), precedence and phase constraints between related processes.

The PMan operation relies on a table known as process record that contains relevant information for each managed process. The record contains information about the process identification, temporal properties (e.g. period, phase), precedence constraints, QoS management and process status. The time management relies on a periodic *tick* controlled by the user, usually generated by a timer or a external event. The process records are saved on table located in a *shared memory* region which access is controlled by *semaphores*, both Inter-process communication (System V IPC) techniques.

3.4.3 Sensorial interpretation, intelligence and coordination

This software component is divided in two applications, the “CAMBADA agent” and the “CAMBADA coach”.

The agent is where sensor-fusion happens and decisions are made. There is the notion of a *world* and its *state* (e.g. distance to the ball, robot location), this is achieved by a function called *Integration* which uses sensor-fusion algorithms to gather the noisy information from the sensors and team mates to update the world state. The high-level function of decision and coordination, based on roles and behaviors [50], operates dependent on the *world state*. The end result is the generation of commands to be sent to low-level control layer.

The coach is the gateway between the agents and human interference. For example, when in competition, the human referee decisions are sent to the coach RTDB and then replicated to the agents.

3.4.4 Vision

Omnidirectional vision

The task of the omnidirectional vision process is to extract information from every frame and relay it to the CAMBADA agent. The vision gathers information about possible ball positions, white lines points and obstacles points both obtained by radial sensors through color and contrast analysis [51]. The information data is then put on the RTDB and afterwards the agent is awaken so that it can processed all the relayed information. Currently, approximately thirty frames are processed per second. Also, it is responsible for initiating PMan.

[1]`http://www.kernel.org`

Frontal vision

The task of the frontal vision process is to detect the balls above ground. For frontal vision tries to gather information about possible ball positions in the air. To reduce the workload the bottom half of the captured frame is discarded. The information data is then put on the RTDB.

The frontal vision is still on an early stage of development, therefore its information is not used for sensor-fusion.

3.4.5 Low-level communication handler

This process is the gateway between the coordination layer and the low-level control layer. Information data from the low-level control layer is read by the handler from the CAN and then put in the RTDB. The same process takes place when information data is to be sent to the low-level control layer, but in reverse order, read from the RTDB and then put in the CAN. The communications requirement are shown in Table 3.1.

ID	Source	Target	Type	T (ms)	Short description
M1	Pic_base	Motor[1:3]	Periodic	30	Aggregate motor set points
M2	Pic_base	Laptop	Sporadic	1000	Battery status
M3.1-M3.3	Motor[1:3]	Pic_odom	Periodic	10	Wheel encoder value
M4.1-M4.2	Pic_odom	Laptop	Periodic	50	Robot position (pos + rot)
M5.1-M5.2	Laptop	Pic_odom	Sporadic	500	Set/reset robot position
M6.1-M6.2	Laptop	Pic_base	Periodic	30	Movement vector (rot + vel)
M7	Laptop	Pic_base	Sporadic	1000	Kicker actuation

Table 3.1: Communication requirements of the lower-level control layer, adapted from [42]

3.4.6 Wireless communication

This process is responsible for broadcasting and updating the shared region of the RTDB. The information is sent through wireless with a period of 100ms, to provide a better QoS it uses an adaptive Time Division Multiple Access (TDMA) protocol [52, 53].

3.4.7 Monitor

This process is a watchdog for the high-level processes. Periodically, it checks the running state of the high-level processes, and in case of abnormal termination they are relaunched.

3.4.8 CambadaConfig

This is an application tool designed to help modify the configuration files used by the high-level software. With it, it is possible to create new formations and simulated them within the tool, set and define new strategies, define the game field dimensions, etc.

3.4.9 Base-station

This application is capable of monitoring and controlling the internal state of the CAMBADA robots. During competition the base-station is responsible to provide automatic processing of the soccer game referring events.

3.5 Summary

At the end of this chapter, one should be familiar with the CAMBADA robots structure and architecture, and the RoboCup project in which the CAMBADA team participates, namely the RoboCup MSL. The different hardware components were presented, including the actuators and sensors. Also, the software architecture and its different components were discussed.

Chapter 4

CAMBADA Simulator

4.1 Introduction

At the beginning of this thesis a survey of existing Robotic Simulators was conducted and presented in Chapter 2. The purpose of the survey was to help us to find a suitable candidate for our requirements or, if there was none, have a general idea of how to create a robotic simulator. Develop a robotic simulator from scratch is a complex and long term project, therefore we looked into the survey and picked one simulator to become our code base.

From the Table 2.1 we can perceive two suitable candidates, SimSpark and Gazebo. SimSpark is known for its use in the RoboCup 3D Soccer Simulation league [26]. Gazebo was also found to be a good choice for the RoboCup MSL [54]. In the end we opted for Gazebo because SimSpark was lacking on viable documentation and the modeling language (RSG) looked harder to understand when compared with Gabzebo's modeling language (XML).

4.2 Simulation environment overview

The CAMBADA team plays in a well defined environment, a MSL field with two goals, a ball and the robots. When transposing to a simulated environment we should start by identifying what is going to be simulated. We started by assuming that any entity with a physical attribute should be simulated. The goals, the ball and the field are all actors in the environment with only physical attributes, the same is not true with the robots, they have components with physical attributes, which we call hardware, and others that only exist as software which may or may not be simulated. In our case we followed the simple premise, all software components that interacts with a hardware component needs to be simulated. The result is the simulation environment depicted in Figure 4.1.

Game environment

Here is where the "Rigid Bodies Dynamics" takes place. The field, goals, ball and robot

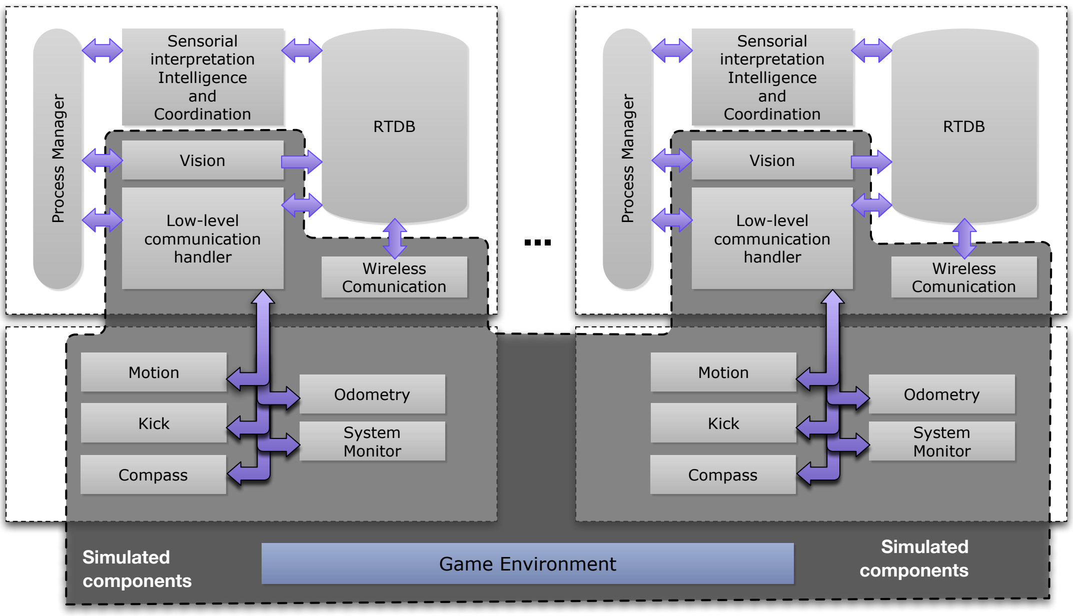


Figure 4.1: Simulated components of the layered software architecture.

bodies are modeled to mimic their real physical attributes such as shape, mass, bounciness and friction.

Sensorial interpretation - Intelligence and Coordination

This a very high level component where the world is interpreted and decisions are made, i.e. *the brain* of the CAMBADA player. This component is what we may call of "testing subject".

RTDB

In the real robot this component serves as middleware for data exchange between components and also between robots. Being a middleware it makes no difference if it is used by real components of by simulated components. From Figure 4.1 we came to conclusion that more than one instance of this component is required for the simulation, which was a problem because multiple instances in the same computer were not supported.

Process Manager

This component fulfills some of the real-time requirements of the CAMBADA software. Much like the RTDB, more than one instance in the same computer of this component was required for the simulation, but such capability was not available.

Vision

The vision gets information from the environment, therefore the information capture by its virtual counterpart is also virtual. The way it extracts the information from the environment should not matter, but it has to be consistent with the information extracted by the real vision.

Low-level communication handler

In the simulated environment there is no communication with real hardware so this component is not necessary in the simulation, but, low-level components still exist. These low-level components need to receive and send information to the components in the higher-level, knowing this, we can assume that in the simulated environment each low-level components is simulated and it has its own *Low-level communication handler*.

Wireless Communication

The goal of this component is to send the shared information in the RTDB to the other robots and receive the shared information of the other robots and put it in the RTDB. Because it runs on a simulated environment the replication of the shared region of the RTDBs does not have to go through a wireless medium before reaching its destiny, we can put it directly in the RTDB of the robots, unfortunately the RTDB did not provide a method to do this.

Motion, Kick, Odometry, System Monitor

These are all low-level components that interact with hardware. In the simulator they are the logic behind the virtual sensors and virtual controllers.

4.3 Necessary changes to non-simulated components

In the previous section (Section 4.2) we gave an overview of the simulated environment and came to conclusion that some of the components that are not simulated required multiple instance in the same computer to allow the simulation to run, namely RTDB (Section 3.4.1) and PMan (Section 3.4.2). Both components rely heavily on System V IPC techniques and were developed with the assumption that they would always be unique, that is, only one instance would exist. The System V IPC techniques in question are *shared memory* and *semaphores*, and each type of resource has an unique identification, usually referred to as key, through the operating system, consequently a different key means distinct shared memory or semaphore resources. The System V IPC resources used by the RTDB and the PMan were created with a pre-defined key. Because the CAMBADA software is replicated to all robots they all used the same key when creating and accessing the System V IPC resources in question, which guaranteed the uniqueness of the components because they were not running in the same computer.

From the Figure 4.1 both RTDB and PMan had an instance for each robot that was being simulated. In the simulation environment these multiple instance ran in the same computer. After the first instance was created the subsequently creations would result in an attach to the already existing instance instead of a new instance. The result was the same instance for all component of the virtual robots.

The challenge was to introduce changes to the RTDB and PMan source code with the minimum impact possible to the components that interact with them, meaning, regardless of the environment in which they run, real or simulated, it should not change the way they are used by the existing components. Although, both components apparently have the same symptoms for not supporting multiple instances the solution is not replicated in both due to different requirements presented by the simulation.

4.3.1 The RTDB problem and solution

The RTDB was already modified to support the simulation environment created by the existing solution, prior the beginning of this thesis. A function was created to initialize the RTDB instances for all agents, and another to free the RTDB instances for all agents. By calling this functions in the simulated environment, the simulated components that use the RTDB interface do not have to worry about creating and freeing the RTDB resources.

To support multiple RTDB instances in the same computer we have to guarantee that the *shared memory* key for each RTDB instance is different, which was not the case. The solution was to extract the agent identification from the environment variable `AGENT` and add it to the pre-defined key. However, in the simulated environment the environment variable `AGENT` would always be the same at each call, therefore for the simulated environment a set of new functions were created that accept the agent identification as parameter. To maintain the interface for the existing components, we replaced the existing functions with wrappers that extract the environment variable `AGENT` and then call the corresponding new function.

When creating a RTDB instance its context is saved in the caller process memory in the form of global variables, which hold the references to the RTDB instance resources. By creating more than one instance in the same process (i.e. the simulator) the context of the previous RTDB instance would be overwritten. To overcome this problem we transformed the global variables into arrays that hold the different RTDB instances contexts. A context is indexed by the agent identification from which the component belonged to.

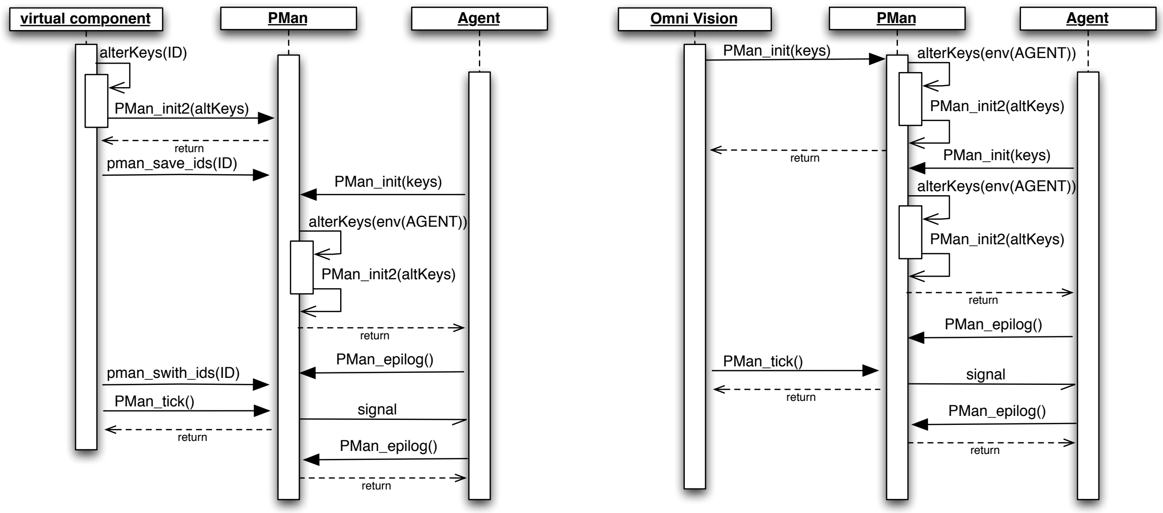
4.3.2 The PMan problem and solution

As initial remarks, a PMan instance is usually referred to as *PMan master* or *master* for short and each master can be identified by the number of the environment variable `AGENT`.

The existing simulation solution, prior the beginning of this thesis, worked without any modification to the PMan and apparently without any problem. While developing the proposed simulator we noticed that all agents were misbehaving, they would start processing at the same time even though only one of them was signaled. We found out that the problem resided in the PMan because there was only one *PMan master* and every *time event* generated would be propagated to all agents. The problem of PMan was the

System V IPC resources, already described. Therefore, to support multiple masters in the simulated environment we had to change the keys that identified the System V IPC resources for each master.

A master is created or attached to by calling the function `PMAN_init()`, some of the arguments passed on to the function is the keys to the System V IPC resources. Theses keys could be changed by the caller, but as already mentioned, introducing changes to other components should be avoided. Although, in the simulated environment a master is created and used by a simulated component that is allowed to change the System V IPC resources keys. The proposed solution is a mixture of multiple masters awareness and function masquerade.



(a) PMan usage on the simulated environment (b) PMan usage on the real environment

Figure 4.2: Sequence diagram of PMan usage.

For the existing components (e.g. agent) we created a new `PMAN_init()` function which will add to the pre-defined keys two times the number found in the environment variable `AGENT`, after that it will call the former `PMAN_init()`, now called `PMAN_init2()`, with the same arguments with the exception of the altered keys. For the simulated components calling the new `PMAN_init()` is not viable because the environment variable `AGENT` would be the same in every call. Although, an identification number with an operational meaning equal to the environment variable `AGENT` is supplied to the simulated component, therefore, before calling `PMAN_init2()`, the key alteration can be made by the simulated component. The proposed changes provides an interface for multiple masters in a simulated environment without affecting the workflow for the existing components, shown in Figure 4.2.

Each time a master was created, its context was saved in the caller process memory in the form of global variables which hold the reference to the System V IPC resources of the master. In the real environment this was not problematic, but, when more than a master was created in the same process (i.e simulator), the previously saved context was overwritten by the context of the newly created master. A simple method to keep the context of a master, when a new one was created, was to save it prior the creation of a new one. Furthermore, the context variables that were used to access the necessary resources to operate, had to be changed accordingly the master in use. This was accomplished by creating a Look-Up-Table (LUT) for each context and use the master identification as key for the LUT. The result was a set of new functions only necessary when running in the simulated environment, namely `pman_save_ids(_master)` after creating a new master, and `pman_switch_id(_master)` before using a specific master.

4.4 Models

Models are physical entities within the simulated environment. The physical aspects of a model contain attributes such as position, orientation, density, joints, etc. A model is build by one or more bodies, each of which is built by one or more geometries or sensors. Controllers can simulate physical devices and can be attached to the model itself or to a sensor.

The physical entities present in our simulated environment are the field, the goals, the ball and the robots. All entities, except the robots, have a very well defined shape and physical behavior without any kind of sensor or controller. The robot model is a little bit more complex, aside from the complex physical structure is has several virtual sensors and virtual controllers that had to be developed from scratch.

4.4.1 The field model

The game field is basically the ground where the ball bounces and rolls and the robot wheels move. We can consider the field markings as just being visual aids and therefore not relevant for field modeling. The field is modeled as being a plane with equation $a * x + b * y + c * z = d$ where (a, b, c), the plane's normal vector, is $(0, 0, 1)$ and $d = 0$.

Considering how the simulator handles the physics parameters of a collision (Section 2.7.2), the physics parameters of the field plane shape are set to high values to allow the other models to define how the collision with the field is handled. The plane is assume to always be a static part of the environment.

4.4.2 The ball model

The ball model is simply a sphere with the same radius and mass as the official FIFA game ball, shown in Figure 4.3. Even though we are simulating rigid body dynamics it is possible to define bounciness parameters to the sphere geometry that will mimic the real behavior of when it collides with other bodies. The bounciness of the ball is defined by the parameters *bounce* and *bouncevel*. The former is a restitution parameter (0..1), 0 means a

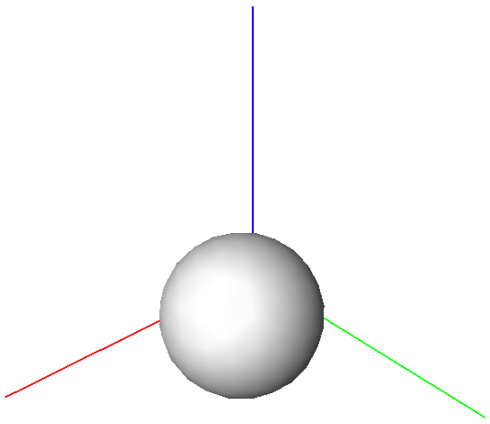


Figure 4.3: The ball model in a perspective view.

surface that does not bounce, and 1 is maximum bounciness. The latter is the minimum income velocity necessary for bounce, any velocity below results in a bounce parameter equal to 0.

To have a bounce behavior similar to the real ball we, adjusted the *bounce* and *bouncevel* parameters by observing how long the ball took to stop bouncing and how many times it bounced when dropped vertically from one meter height. Afterwards, we repeated the experiment in the simulator tweaking *bounce* and *bouncevel* until we got a similar behavior.

4.4.3 The goal model

The goal forms a space over which the ball has to be sent in order to score and for the much frequent ball on the post or crossbar. The goal is modeled with the dimensions defined in the RoboCup MSL rules. Basically a goal is four rectangular cuboids displaced like the ones shown in Figure 4.4. A goal is also a static part of the environment.

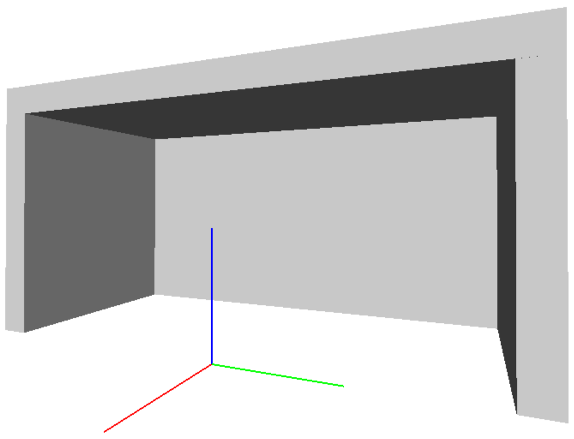


Figure 4.4: The goal model in a perspective view.

4.4.4 The robot model

The robot model is the most complex one. As already mentioned, we have to model the robot physical shape and also its sensors and actuators (i.e. controllers). Here we will only discuss the modeling of the robot physical shape, the virtual sensors and virtual controllers will be discussed ahead with more detail.

From Section 3.3.1 we perceive that the physical shape of the robot can be modeled to resemble a cylinder and a cone on top. To simplify we discarded the top conic shape, which represent the vision set up, and also all physical components with movable part, i.e. kicker and grabber. To recreate a cylinder with the cut out section that keeps the ball under control we created an outer case composed by small rectangular cuboids displaced along the robot perimeter except at the front of the robot. Inside the case we placed a cylinder with a smaller radius but with the same height. This latter cylinder will simulate most of robot weight and will stop the ball from entering too much inside the case.

The holonomic motion of the robot also has to be modeled, meaning, the virtual robot should move by the means of three swedish wheels with the same disposition of the real robot (Section 3.3.2). The holonomic motion system is modeled by three spheres connected to the robot's chassis by hinge joints acting is if they where the motors. The joints will keep the wheels and the chassis together and will also be used to apply angular velocity to the wheels. Modeling a swedish wheel with a sphere is a viable option because it is assumed that the contact between the wheel and the ground is reduced to a single point on the plane [47].

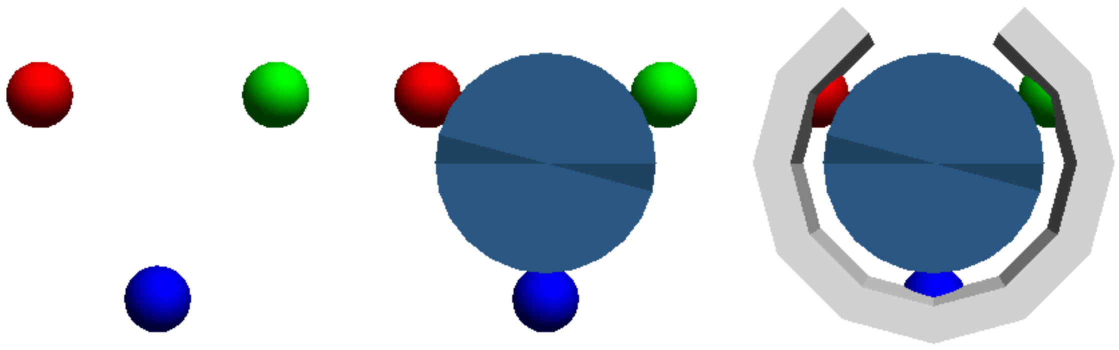

(a) Stage 1 - The wheels (b) Stage 2 - The inner cylinder (c) Stage 3 - The outer case

Figure 4.5: The robot model assembling.

In summary, the robot model is assembled by three parts: the wheels, the inner cylinder and the outer case, shown in Figure 4.5. The Figure 4.6 gives a perspective view of the robot model.

Applying the right angular velocities to the joints did not sufficed to emulate the holonomic motion of the robot. We also had to set the Coulomb friction coefficient and the force-dependent-slip coefficient (FDS), both physics parameter to handle collisions. The friction coefficient define the friction between the wheel geometry and the field geometry in the perpendicular direction of the wheels rotation axis. The FDS coefficient define how much the wheel geometry will slip along the parallel direction of the wheels axis.

To adjust these parameter we looked into the source code of a more experimented simulator, namely *RoboCup Simulator Dev* (Section 2.4). *RoboCup Simulator Dev* also used ODE for its dynamics simulation, and the physics parameters that help emulate the holomic motion of the robot can be transposed to our simulation.

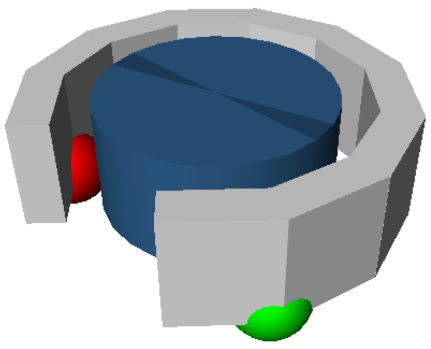


Figure 4.6: The robot model in a perspective view.

4.4.5 The obstacle model

The obstacle model is a simplified version of the robot model. It does not have the wheels, the inner cylinder is wider and the outer case is closed (Figure 4.7). In practice, the obstacle model can be used to represent an adversary.

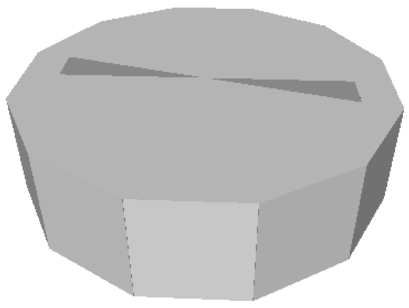


Figure 4.7: The obstacle model.

4.5 Sensors

A virtual sensor does not have a physical representation but it creates and returns information from the simulated-world. A virtual sensor is a C++ class instantiated by the

"SensorFactory", and has to be attached to a body. Every virtual sensor has an update rate that defines how periodically the sensor will update its information data.

For the proposed simulation environment we developed a virtual omnidirectional vision, a virtual compass and a virtual barrier. A virtual frontal vision was not developed because its data is yet to be use by the *Sensorial interpretation, intelligence and coordination* component.

4.5.1 Omnidirectional Vision

We already know what information the vision gathers (Section 3.4.4), now, we have to transpose it to a simulated environment. To the agent the origin and how the vision data is obtained does not matter, the RTDB makes it transparent, therefore the simulated vision does not have to extract the information from the environment the same way as the real vision. As long as the type of data and operational workflow (e.g. awake the agent, delay) are kept the agent will consume the data without caring about its provenance. Although we should try to mimic the process of data acquisition as much as possible.

Due to the nature of a simulated environment, every entity in the simulation (e.g. the ball model) is always accessible and detectable, which is not true in the real world. Sometimes the ball can fall behind other robots and become undetectable, the same can happen to the white lines used for self-localization. To increase the accuracy of the simulation we have to simulate this phenomenon which we call *occlusion*.

The simulated vision workflow, depicted in Figure 4.8, will be discussed in the sections that follow. As remark, *Detect white points*, *Detect ball* and *Detect obstacles* operations are depicted as being running in parallel, which is not accurate. They run sequentially, but because they are independent from each other the order in which they run does not matter, therefore they are depicted as being parallel.

Fill obstacle list

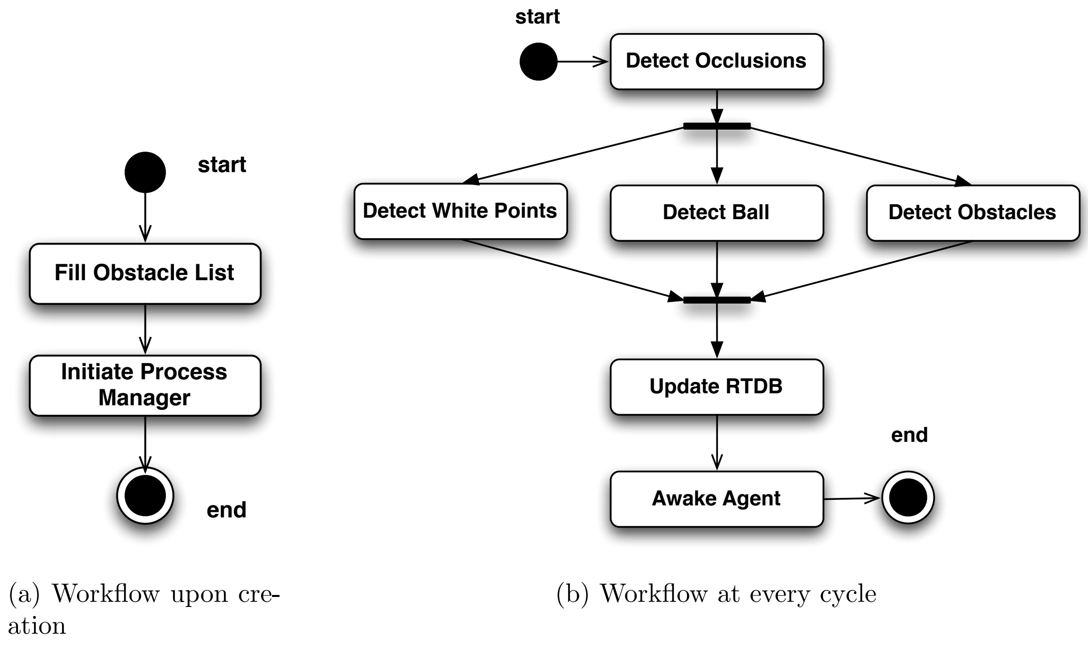
The structured environment of the RoboCupMSL enforces the color black as a robot, or in a more abstract sense, an obstacle. The vision sensor queries all models in the World and the ones with black color are added to its internal list of obstacles. This list is filled only once because we make the assumption that once the simulation starts objects are not added or removed dynamically.

Initiate Process Manager

The vision process has the responsibility of initiating PMan, thus to keep the original operational workflow the simulated vision has to do the same.

Detect Occlusions

To simulate occlusion in the 3D environment of the simulator, we start by projecting the world to a *top view*, similar to the reverse-mapping into distance map [55] used by the



(a) Workflow upon creation

(b) Workflow at every cycle

Figure 4.8: Simulated omnidirectional vision workflow.

real vision. From this view we can construct what we call **occlusion area**. The *top view* is easily understood as the XOY plane.

Because we are simulating a SVP vision the obstacle creates an occlusion area that is limited by two lines, each one tangent to one side of the obstacle and starting at the obstacle position. To simplify the process we made the assumption that the obstacle is a cylinder with the same height as the robot that hold the vision, although we made no assumption about its radius, therefore we get the radius from its Axis-Aligned Bounding Box (AABB). The assumption is viable because the obstacles most likely are robots from our team or the opposing team and also because assuming the obstacle is a cylinder we can easily calculate the tangents with Thales' Theorem [56].

From the *top view*, depicted in Figure 4.9, we can obtain the tangents with Thales' Theorem. By constructing a circle centered at M through V and O, the point T is the intersection of this circle with the given circle of the obstacle, because that is the point on the circle of the obstacle that completes a right triangle VOT. It is easy to perceive that the occlusion area can be limited by an area between the angles made by T and T' with the robot and starts beyond the obstacle. The resulting occlusion area, by it self, is enough for points at ground level but not otherwise. As already stated, we made the assumption that the obstacles are of cylindrical shape and have the same height as the robot holding the vision, because of this, in our vision set up we can assume that every occlusion area extends to the horizon making the occlusion area extracted from the *top view* enough for all dimensions.

This method is applied to every obstacle in the list (Section 4.5.1) and the resulting

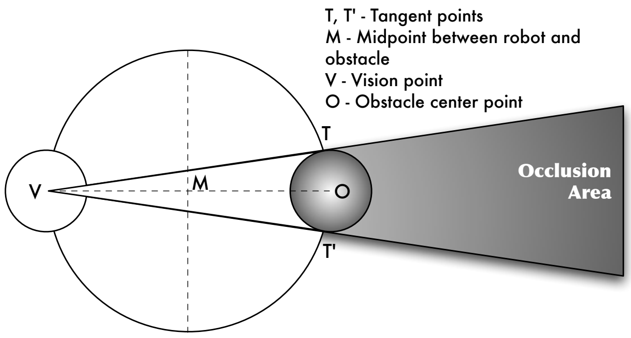

T, T' - Tangent points
M - Midpoint between robot and obstacle
V - Vision point
O - Obstacle center point
T
V
M
O
T'
Occlusion Area

Figure 4.9: Detection of occlusion area with a top view.

occlusion areas saved to be queried later.

Detect Obstacles

The simulated vision searches for obstacles points with an algorithm similar to the one implemented in the vision[51]. It uses radial sensors to search for obstacles but instead of looking for the transitions between pixels [57] it will cast rays to intersect with the obstacles. Just like in the detection of occlusions the world is projected to a *top view*. The Algorithm 1 shows the pseudo code of obstacle point detection.

At the beginning the step angle is fixed to 1.5 degrees which results in 240 radial sensors. The main cycle will end at $\alpha \geq \pi$ because the line that are used to intersect with the obstacles circle are of infinite length and due to inverse symmetry at the origin it will cover intersections for $\pi < \alpha \leq 0$ and $-\pi \leq \alpha < 0$. The function *radiusFromAABB* will evaluate the obstacle AABB to find the biggest distance between the maximum and minimum components, the resulting distance can be seen as the diameter of the obstacle. The *line* will act as a radial sensor and the *circle* as an obstacle in the *intersectAnGetClosestPoint* function. Because in line intersection with circle there can be two points, only the closer to the origin is return. To have a more accurate simulation, the obtained points are check against all occlusion areas to see if they are visible, if they are they are kept.

Ball detection

What is inserted in the RTDB by the real omnidirectional vision is the distance between the robot and the lowest point of the detected ball and has a limit of how far it can detect a ball, so we should try to do the same.

The trivial case in when the ball is on the floor. In this case the information extracted from the environment is the distance between the robot and the ball. When the ball is above ground we try to mimic the effect introduce by the hyperbolic catadioptric vision system of the ball appearing to be further then it really is. To simplify the process, when the ball is above ground, we project its position onto the ground, shown in Figure 4.10.

Algorithm 1 Obstacle detection with line intersection

```
$\alpha \leftarrow,\quad \theta \leftarrow \frac{2\pi}{240},\quad blackpoints \leftarrow empty0$
while $\alpha < \pi$ do
  $line \leftarrow y = \frac{\sin\alpha}{\cos\alpha}x$
  $points \leftarrow empty$
  for all $obstacles$ do
    $radius \leftarrow$ radiusFromAABB($obstacle$)
    $(a, b) \leftarrow$ relativePosition($obstacle$)
    $circle \leftarrow (x-a)^2 + (y-b)^2 = radius^2$
    $point \leftarrow$ intersectAnGetClosestPoint($line$, $circle$)
    $points \leftarrow$ add $point$
  end for
  for all $points$ do
    if $point$ not in occlusionArea then
      $blackpoints \leftarrow$ add $point$
    end if
  end for
  $\alpha \leftarrow \alpha + \theta$
end while
return $blackpoints$
```

From the projection we get:

$$\tan\alpha = \frac{h}{d} \wedge \tan\alpha' = \frac{h-h'}{d'} \wedge \alpha = \alpha'$$

$$\implies \frac{h}{d} = \frac{h-h'}{d'} \iff d = \frac{hd'}{h-h'} \tag{4.1}$$

From Equation 4.1 we get the distance d between the robot and the projected ball, which will eventually be the detected distance to the ball. This method can also be used in the trivial case where $h' = 0 \implies d = d'$, thus covering the detection for both cases.

For the vision to successful detect a ball, first it must not be beyond the vision *see* limit and then not be in an occlusion area. In the first case the inequality $h' < -\frac{hd'}{s} + h$, where s is the maximum *see* distance, tell us if the ball is detectable and for the latter we have to check if the real position of the ball is not on a occlusion area, previously detected in Section 4.5.1. The ball is considered to be a particle and not a blob, therefore partial occlusion of the ball is not implemented.

White points detection

From the RoboCup MSL rules the lines that delimit the game field areas are white and these are used by the robots for self-localization purpose. Much like for object points

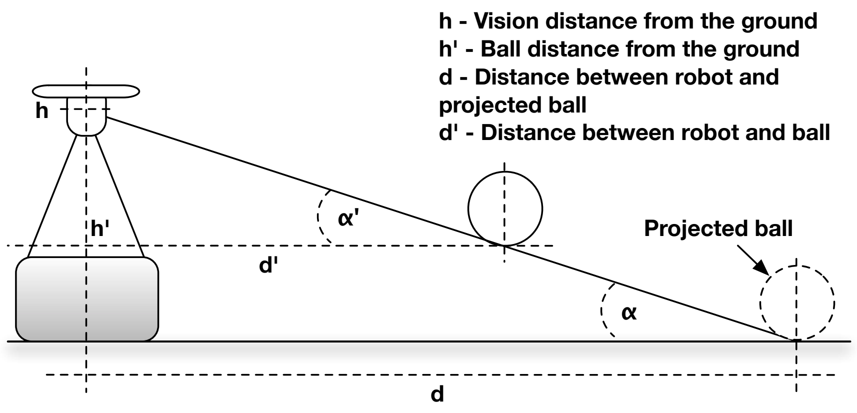

h - Vision distance from the ground
h' - Ball distance from the ground
d - Distance between robot and projected ball
d' - Distance between robot and ball
h
h'
α'
d'
Projected ball
α
d

Figure 4.10: Ball detection above ground - Projection.

detection (Section 4.5.1), the vision makes use of radial sensors. Once again there is no actual image to look for transitions between pixels, thus another method has to be used. If we represent the field as a conjunction of line segments we can easily use line intersections to obtain virtual white points. Again the world is projected to a *top view*.

The number of white points that will be extracted from the environment is not known but there is a ceiling for how many points the vision will transmit. If we took the traditional way of going through all the sensor in a consecutive order, then the points number ceiling will be rapidly attained and will result in areas left out for white point detection. To attenuate this effect we go through the sensors in multiple passages. At each passage only a subset of the radial sensors are used. Each subset only contains radial sensors separated apart by a number os sensors equal to the number os passages. The Algorithm 2 shows the pseudo code for white point detection.

The function *fieldIntersect* will use the given line for intersection against all the line segments that delimits the field areas and returns the intersection points. Each received points is checked for occlusion and discarded if true.

Update RTDB, Awake Agent

The vision and the agent work synchronously where the vision precedes the agent in execution. When the vision reached the end of its processing cycle it signals the agent that new data is ready to be processed. In loose terms the agent awakes, grabs the vision data from the RTDB, executes its cycle and then goes to sleep to be awaken again by the vision. This is a very important step in the vision workflow because without awaking the agent the system would stall.

Algorithm 2 White line detection with line intersection

```
  $\theta \leftarrow \frac{\pi}{sensors}, whitepoints \leftarrow empty$
  for $pass = 0$ to $radialPassages$ do
    $\alpha \leftarrow \theta * pass$
    while $\alpha < (\pi - \frac{\theta}{2})$ do
      $line \leftarrow y = \frac{\sin\alpha}{\cos\alpha}x$
      $points \leftarrow$ fieldIntersect($line$)
      for all $points$ do
        if $point$ not in occlusionArea then
          $whitepoints \leftarrow$ add $point$
Ensure:     totalOfWhitePoints $\leq$ maxWhitePoint
        end if
      end for
      $\alpha \leftarrow \alpha + \theta * radialPassages$
    end while
  end for
  $whitepoints \leftarrow$ convertToRelativePosition($whitepoints$)
  return $whitepoints$
```

4.5.2 Compass

The purpose of this sensor is very simple, give the yaw angle of the robot frame. Because we are on a simulator this kind of information is easily obtain by querying the robot body about its rotational state. The compass is a low level component, thus its data is handle by the *Low-level communication handler* component, but in the simulator we can ignore this middleware and put the data directly in the RTDB.

4.5.3 Barrier

The barrier sensor serves as a signaling flag to when the ball is under control. The cut out section of the robot body is where the ball is kept under control. The real barrier sensor is implemented with an infrared circuit strategically installed at the bottom of the lower layer. When the circuit is interrupted it means that the ball is on the control handling area. It is possible to do this kind of sensors in the simulator but we took a more analytic approach. Being on a simulated environment the sensor can know the exact distance between the ball and the robot and taking the form of the robot into account it can accurately pinpoint when the ball is under control. The *control area* can be delimited by polar coordinates relative to the robot frame, where α is the maximum absolute polar angle and d the maximum radial coordinate. For the ball to be on the *control are* its distance d' to the robot and the angle θ it makes with it, must conform with $|\theta| \leq \alpha \wedge d' \leq d$. Another condition is that the ball must must be on the ground or slightly above. The example in Figure 4.11 shows a *control area* delimited by d, with the same length as the robot radius, and α. In the example the ball is slightly on the *control area* which is enough to assume the ball is under control. The parameter d was calculated by measuring the distance from the center of the robot to end tip of the grabber wheel (Figure 3.7c).

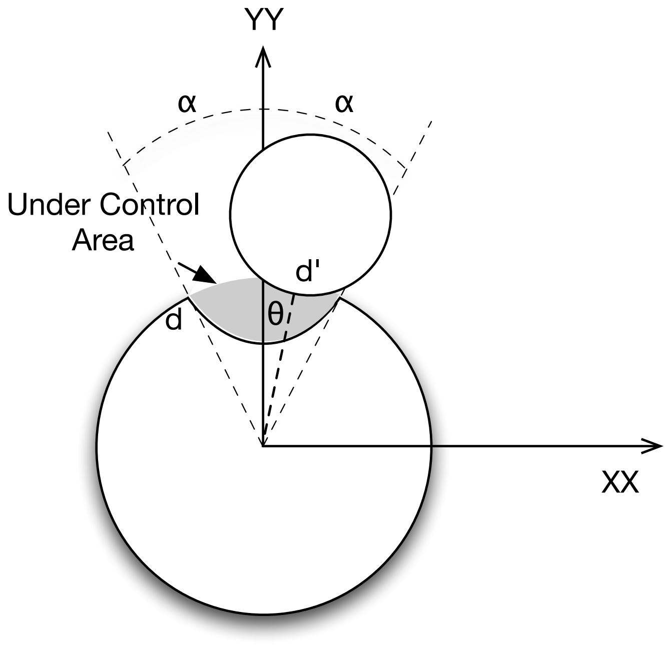


Figure 4.11: Under Control Area defined by the barrier sensor.

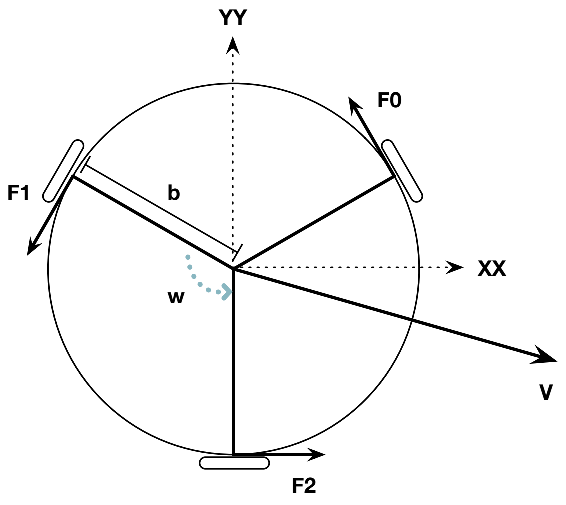
4.6 Controllers

A virtual controller is meant to control joints and simulate non existing physical structures (e.g. the kicker system). A virtual controller is a C++ class instantiated by the "ControllerFactory", and can be parentless or be attached to a sensor. Every virtual controller has an update rate that defines how periodically the controller will actuate.

For the proposed simulation environment we developed a holonomic motion system, a grabber, a kicker system and a wireless communication module.

4.6.1 Holonomic motion

The virtual holonomic motion controller works intrinsically with the joints of the holonomic motion system modeled in Section 4.4.4. As mentioned, the joint is the virtual counterpart of the motor, therefore, angular velocity is applied to the joint and, due to the constraints created by the joint, the body of the wheel connected to that joint will also rotate. Inverse kinematics is used to translate the desired velocities to apply at the robot frame into the wheels angular velocitiy and can be deduced from the constraints of the holonomic motion setup [47] [58], shown in Figure 4.12.



- $n \equiv$ Wheel number
- $r \equiv$ Wheel radius
- $F_n \equiv$ Unit direction vector
- $V \equiv$ Linear velocity of the body frame
- $w \equiv$ Angular velocity
- $b \equiv$ Wheel baseline
- $r \equiv$ Wheel radius

$$F_0 = [-\frac{1}{2}, \frac{\sqrt{3}}{2}] \quad F_1 = [-\frac{1}{2}, -\frac{\sqrt{3}}{2}] \quad F_2 = [1, 0]$$

Figure 4.12: Constraints of the holonomic motion system.

The velocity direction is constrained by each wheel. At each wheel n, the velocity P_n depends on V and w, and is the sum of the body frame linear velocity V and the angular velocity w:

$$P_n = V + (b \cdot w) * F_n$$

And $P_n \cdot F_n$ is equal to the wheel n velocity v_n, therefore:

$$v_n = V \cdot F_n + b * w$$

Now, we can calculate the angular velocity w_n for each wheel n:

$$w_n = \frac{v_n}{r} = \frac{V \cdot F_n + b * w}{r}$$

$$\begin{aligned} w_0 &= \left(-\tfrac{V_x}{2} + \tfrac{\sqrt{3}V_y}{2} + b * w\right)/r \\ w_1 &= \left(-\tfrac{V_x}{2} - \tfrac{\sqrt{3}V_y}{2} + b * w\right)/r \\ w_2 &= \left(V_x + b * w\right)/r \end{aligned} \tag{4.2}$$

The coordination layer generates the desired linear velocity V_x, V_y and angular velocity w to be applied to the robot frame and puts it in the RTDB, and, before it is sent to the Motion function at the low-level control, the *Low-level communication handler* (Section 3.4.5) translates the desired linear and angular velocity of the robot frame to the wheels setpoints. The mentioned "translation'" is handled by a *Holonomic Motion* [59] module. Because this is a virtual controller the *Low-level communication handler* is within itself, therefore the very same *Holonomic Motion* module is used in the this controller to translate the "orders from above".

The *Holonomic Motion* module accepts as input parameters the velocities V_x, V_y and rotation V_a, and return the three wheels setpoints. The wheels setpoints were obtained by multiplying the calculated w_n with the *wheel setpoint factor*. Therefore, to apply the correct angular velocities to the wheels joints we had to multiply the wheels setpoints with the inverse *wheel setpoint factor*.

The data generation of the odometry (Section 3.3.3) is implemented in this controller. Knowing that the outputted data is the displacement of the robot frame and its rotation, the displacement O_x, O_y and rotation O_a can be calculated with an Euler integration:

$$O_x = O_x + (\cos O_a v_x - \sin O_a v_y)\Delta t \quad O_y = O_y + (\sin O_a v_x + \cos O_a v_y)\Delta t \quad O_a = O_a + v_a \Delta t$$

The variable Δt is the amount of time in which the robot frame had the velocities v_x, v_y and ω. Although we received the desired v_x, v_y and v_a from the coordination layer, they may be changed by the *Holonomic Motion* module that applies saturation and acceleration filters to v_x, v_y and v_a. Fortunately the altered v_x, v_y and v_a are saved in the *Holonomic Motion* module, therefore we can use them in the calculation of O_x, O_y and O_a.

4.6.2 Grabber

For the grabber (Section 3.3.6) we did not create a virtual counterpart with a physical attribute. The grabber has several moving parts and could be difficult to model, therefore we decided to create a virtual grabber using only the logic behind it. The goal of the grabber is simple, keep or bring the ball under control, for that the ball is rotated towards the cut out section in the front part of the robot. The grabber is controlled by a power value which translates in how much spin the grabber's wheels has. When the ball is on the *control area*, which is signal by the barrier sensor, the wheel of the grabber is disabled by the coordination level, i.e. power is set to zero.

This virtual controller has a method of operation similar to the virtual barrier sensor (Section 4.5.3) to know when the ball is in the range of its influence. To compensate the non existence of the grabber's wheel the virtual controller, when the ball is range, applies a force vector at the ball towards the center of the robot. This force we call it "attraction force", as it will attract the ball to the *control area*.

4.6.3 Kicker

The kicker (Section 3.3.7) can perform two types of kicks, direct and lob kick (Figure 4.13). As mentioned, the granular control of the device power results in a controllable initial velocity to when the ball is kicked. What differentiates the direct kick from the lob kick is the angle at which the ball is hit. On a direct kick the ball is hit at an approximate angle of 0 degrees and is expected to travel in parallel with the ground on a straight line. A lob kick, like the name implies, throws the ball in a high arc which is achieved by hitting the the ball at an approximate angle of 45 degrees. At a lower level, the kicker is controlled by a power value, which is an eight bit number sent by the coordination layer. The kick type is differentiated by the most significant bit of the power value, 0 means lob kick (high kick) and 1 direct kick (low kick). The power value is capped to 50. This controller follows the same method of operation of the kicker here described.

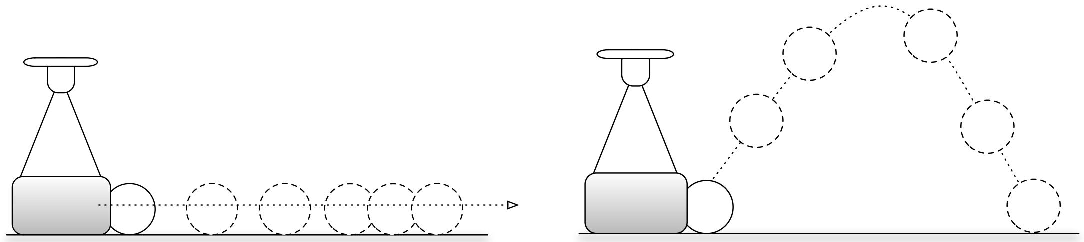

(a) Direct kick (b) Lob kick

Figure 4.13: Type of kicks created by the kicking device.

High kick

It is easy to perceive that once the ball leaves the ground it becomes a projectile drawing the path of a projectile. The motion of a projectile in a three dimensional environment is something that our *Rigid Body Dynamics* can and will simulate, it is just a matter of applying the right velocity to the virtual ball. To find the velocity vector to be applied at the virtual ball we looked for a system with a similar behavior, a cannon and its cannonball [60].

The first step is to specify the coordinate system and the cannon location, for that we use a right-handed coordinate system. The butt end of the cannon shaft is located at the origin and its tip is calculated with the cannon length, the "tilt" angle and "swivel" angle. The cannon length L is cannon's shaft length and it can be an arbitrary value greater than zero, the swivel angle δ is the away rotation of the cannon from from the vertical plane and the tilt angle θ is the upward rotation from the horizontal plane (Figure 4.14). The tip of the cannon is calculated as follow:

$$tip_x = L\cos\theta\cos\delta \quad tip_y = L\cos\theta\sin\delta \quad tip_z = L\sin\theta$$

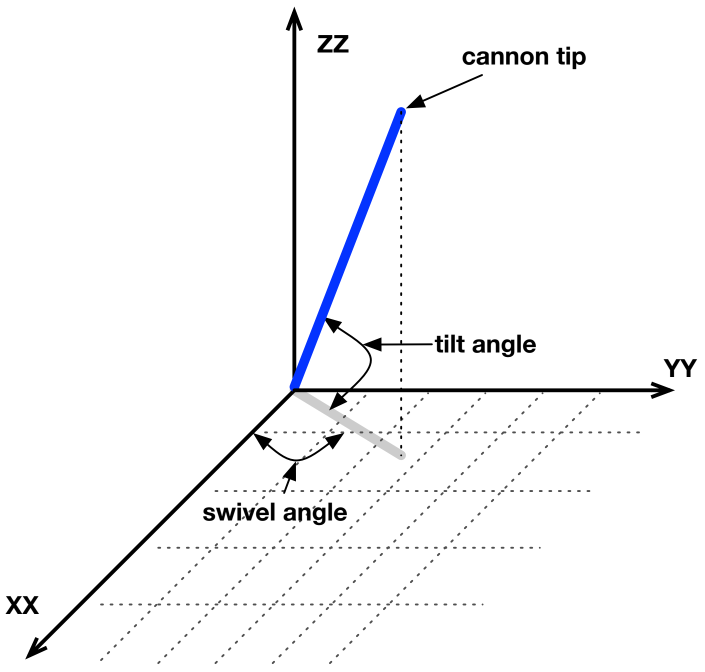


Figure 4.14: Reference coordinate system and cannon location.

The next step is to find the normalize direction vector:

$$dc = \begin{bmatrix} \frac{tip_x}{L} \\ \frac{tip_y}{L} \\ \frac{tip_z}{L} \end{bmatrix}$$

The final step is to find the initial velocity of the projectile. In the coordination layer the lob kick is resolved in term of range and then converted to a power value, which is what this controller reads. Instead of reconverting the power value to a range it is converted to an initial velocity v_i, which is more useful for the projectile modeling. The conversion is the product of the normalized power value with the maximum initial velocity imposed by the virtual kicker:

$$v_i = v_m \frac{p}{p_m} \tag{4.3}$$

The power value p is capped to p_m. The maximum initial velocity v_m should be somewhat approximate to the real initial maximum velocity. From the real kicking device we can measure its maximum range r_m and also its tilt angle θ. From the projectile trajectory equations and ignoring drag, the maximum velocity v_m can be obtain from r_m and θ:

$$r_m = \frac{v_m^2 \sin 2\theta}{g} \qquad \Longrightarrow \qquad v_m = \sqrt{\frac{r_m g}{\sin 2\theta}} \tag{4.4}$$

The velocity v to applied to the projectile is obtain by multiplying dc with v_i:

$$v = \begin{bmatrix} \cos\theta\cos\delta\sqrt{\frac{r_m g}{\sin 2\theta_i}}\frac{p}{p_m} \\ \cos\theta\sin\delta\sqrt{\frac{r_m g}{\sin 2\theta_i}}\frac{p}{p_m} \\ \sin\theta\sqrt{\frac{r_m g}{\sin 2\theta_i}}\frac{p}{p_m} \end{bmatrix} \tag{4.5}$$

The end result is a vector v that depends on the θ, δ, p and r_m. The values of θ and r_m are static they are provided when the controller is initiated and kept through the simulation. The others variable values are dynamic, δ is the orientation of the robot front and p is the power value excluding the kick type selection bit.

The value of θ was calculated by analyzing the video of a robot kicking the ball with a high-kick. The value of r_m was obtained by kicking the ball with a high-kick at maximum power, and measuring the distance between the point of origin of the ball and the location were the ball touch the ground for the first time.

Low kick

The difference from the high kick is the angle at which the ball is hit, which is zero degrees because the ball is kicked parallel to the ground. Also, we can assume that the maximum velocity v_m that the kicker system applies in the low kick to the ball model is the same in Equation 4.4, and the power conversion v_i is equal to Equation 4.3. Therefore, the velocity vector to be applied to the ball model can be calculated by multiplying the direction vector unit with v_m and v_i:

$$v = \begin{bmatrix} \cos\delta \\ \sin\delta \\ 0 \end{bmatrix} v_m v_i$$

The end result is a vector v that depends on θ which is the virtual robot orientation, and v_m that is calculated when the controller is initiated.

4.6.4 Wireless communications

The wireless communication process (Section 3.4.6) transmits its data through a wireless medium which does not exist in the simulated environment. To simulate the replication of the shared region of all robots RTDB, we created a virtual controller that has read and write access to all robots shared region of the RTDB, therefore instead of sending the data through a medium it is inserted in the RTDB. This controller does not need to be associated to a virtual robot because it already does the replication for all RTDB instances. Therefore, the controller enforces its attachment to an empty model (i.e without any physical representation) and prevents unnecessary multiple instances by allowing only one instantiation. This controller also initiates an RTDB instance for each agent, and also terminates all RTDB instances when the simulations ends.

4.7 Visualization and interaction

As mentioned in Section 2.7.1, the visualization engine of Gazebo has a GUI and a three-dimensional rendering engine. The GUI provides interaction with the simulation while the rendering engine draws the simulated environment. The used rendering engine, OGRE, allow the development of high-end application utilizing hardware-accelerated 3D graphics. The problem with OGRE was that it required hardware capable of supporting high-graphics, which did not complied with the low-graphics capability requirement. Therefore, we made the decision to remove OGRE and the GUI, from the code base, and create a new visualization engine with a simpler rendering engine and GUI.

4.7.1 GUI

The GUI was developed using Qt4[1] libraries. It serves as a placeholder for the rendering engine widget and handles mouse and keyboard events.

The GUI offers interaction with the rendering point-of-view and also interaction with the simulation. For the rendering point-of-view, the GUI offers four pre-defined views:

Top view present an orthogonal view of the game field from a point above the center of the field (Figure 4.15a);

Ball view shows a top view while tracking the virtual ball (Figure 4.15b);

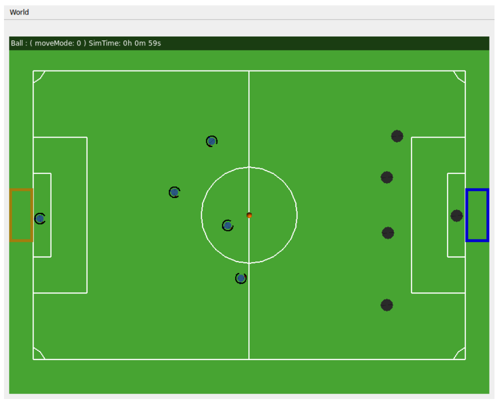
Free view allows the user to manipulate the rendering point-of-view with the mouse (Figure 4.15c);

FRB view is a “first robot view” that renders the world from a robot point-of-view (Figure 4.15d). This view is experimental, therefore yet limited to a specific robot.

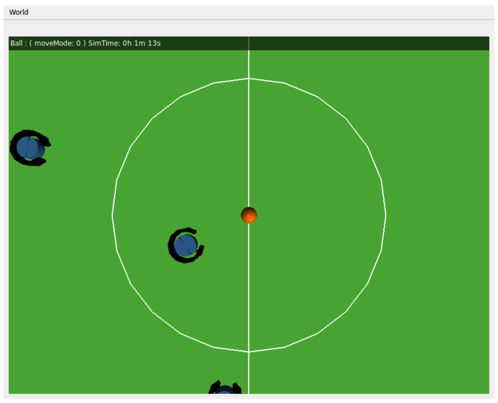
With the GUI it is possible to save and restore the state of each model present in the simulation at any given time. A model state is represented by its pose in space, linear velocity, linear acceleration, angular velocity and angular acceleration. The world state resides in the process memory, and when it is saved, it will overwrite the previous saved state. Also, the virtual ball position can be controlled by the keyboard.

When the **Top view** is active the GUI offers another interaction with the simulation. Within the rendering widget it is possible to change the location and orientation of any non static model in the simulation by just dragging-and-dropping the selected model (Figure 4.16). The drag-and-drop interaction has three steps, selection, dragging and dropping. To select a model the user clicks on the view with his mouse, the click coordinates in the

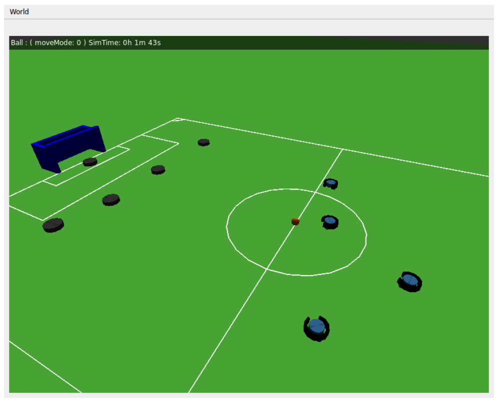
[1]`http://qt.nokia.com/`



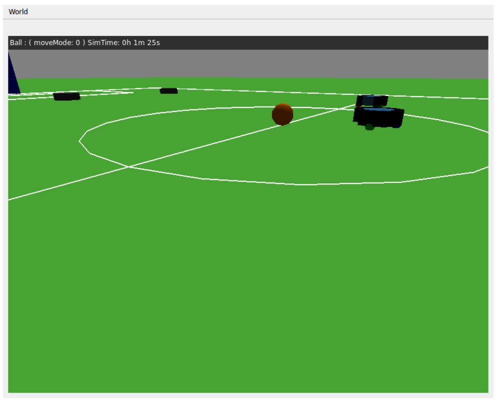
(a) Top view



(b) Ball view



(c) Free view



(d) FRV view

Figure 4.15: The four views of the rendered world.

rendering widget are translated to world coordinates, afterwards the world is queried for a model in that world coordinates. If a model exists in that coordinates then it is selected and the drag and drop activated. The dragging of a selected model can calculate its new location or orientation depending if it was selected with the left button (location) or with the right button (orientation). For visual aid, while dragging, a line is drawn from the selected point to the current point of the mouse pointer and new values are printed in the left top corner. On drop, the location or orientation of the model is changed. The new location is obtained by translating the drop position to world coordinates. The new orientation is the polar angular of the resulting pole from the select point to the drop point. Any change to the simulation happens in real time, meaning, it will affect immediately the simulation without the need to stop and restart the simulation. However, selecting the ball model with the right button is a special case (Figure 4.16c). Instead of calculating a new orientation and applying the change to the simulation, it calculates a new velocity and saves the current state of the ball model with the calculated velocity, therefore the new velocity only takes place when the world state is restored. The new velocity is the resulting vector from the select point to the drop point.

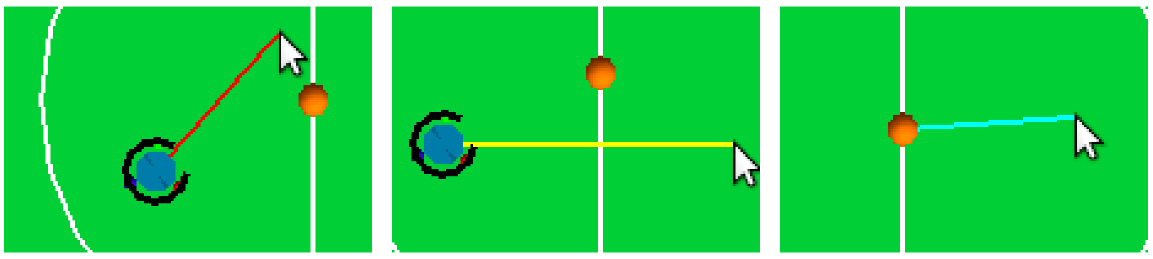

(a) Model position (b) Model orientation (c) Ball velocity

Figure 4.16: Example of drag-and-drop interaction with the simulation.

4.7.2 Rendering engine

The rendering engine borrows its rendering primitives from the library “drawstuff” that is included with ODE. Due to the simplicity of the rendering engine, it will only render the geometries that build up the bodies of the defined models in the simulation, i.e only physics geometries are rendered (e.g spheres, boxes, cylinders). The end result was a lightweight rendering engine capable of performing in computers with low-graphics capabilities, which complies with our operational requirements.

4.8 Summary

In this chapter we present the developed simulation environment for a robotic soccer agent as well the integration of the CAMBADA software into the new simulation environment. We gave a more detailed description of the physics and visualization engine of the developed simulator. We discussed the insights of our simulation environment and the existing components of the CAMBADA software. Furthermore, we discussed the modeling of the robots and their environment, and the development of the controllers and sensors virtual counterparts. Finally, we presented the simulator GUI, that holds the rendering widget and provides interaction with the simulation, and the rendering engine that draws the world.

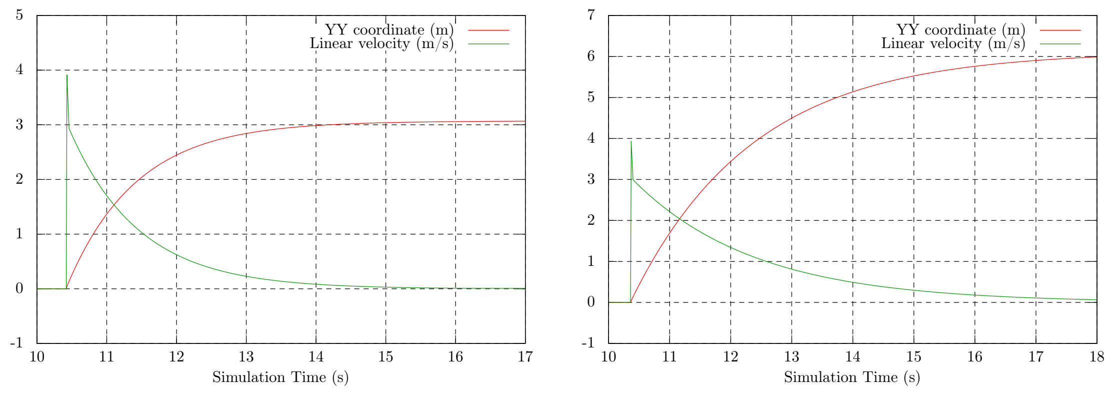
Chapter 5

Results and Troubleshooting

5.1 Ball motion

The ball model has two characteristics that we should consider: the rolling friction, and how much it will bounce upon collision. The first should be considered by acknowledging that ODE does not model rolling friction, which results in the ball rolling off into infinity. Applying damping to the moving ball will prevent this behavior After the ODE stepper function, linear and angular velocities are tested against the corresponding threshold. If they are above they are multiplied by $(1 - d_c)$. The damping coefficient d_c can be tweaked to slow down the ball at a desired rate. A high damping coefficient can make the ball stop rolling at a higher rate, but it will affect the expected motion of the ball when it becomes a projectile.

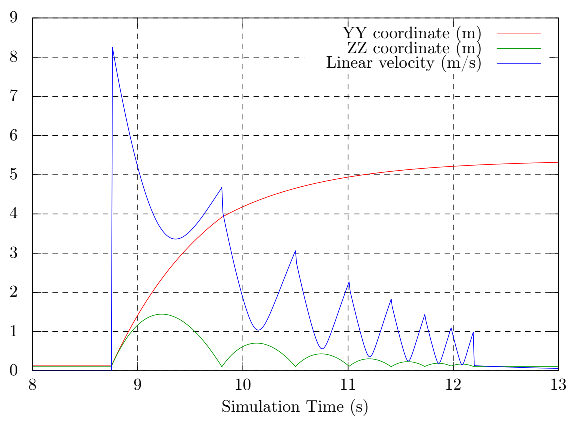
The damping effect on the ball rolling along the YY axis is depicted in Figure 5.1. It shows that the damping coefficient affects the ball linear velocity. The lower the damping coefficient, the lower the slowdown rate of the ball.



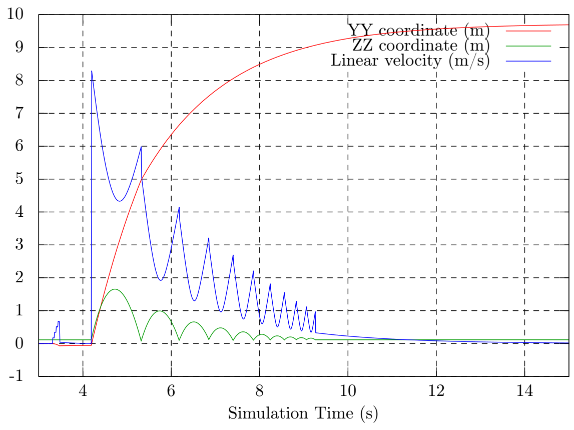
(a) With damping coefficient equal to 0.01. (b) With damping coefficient equal to 0.005.

Figure 5.1: Damping effect on the ball rolling.

The damping effect on the ball when it is thrown into the air along the YY axis is depicted in figure Figure 5.2. It shows that decreasing the damping coefficient results in a larger distance traveled in a similar time span before the ball bounces for the first time. It also results in an increase of the number of times the ball bounces, and a larger time span before the ball stops moving. The reached height by the ball appears to be similar, meaning, the damping coefficient affects the distance traveled by a projectile but not the "flight" time. By consequence the bounce of the ball is also affected by the damping coefficient.



(a) With damping coefficient equal to 0.01.



(b) With damping coefficient equal to 0.005.

Figure 5.2: Damping effect on the ball when its thrown into the air.

The bounce that results from the virtual ball hitting the game field is important because it is taken into account when the agent issues a high kick. From the method presented in Section 4.4.2 we extracted the values that help characterize the bounciness of the ball. From the repeatedly carried experiment, the real ball would stop after six bounces in 2.81 seconds (mean). After empirically adjusting the bounciness parameters we obtained a similar bounce effect in the simulator. The virtual ball was dropped at one meter high and bounced six times before it came to rest after approximately 2.87 seconds (Figure 5.3).

5.2 Sensors and controllers

5.2.1 Omnidirectional vision

The information generated by the virtual omnidirectional vision has direct effect on the agent self-localization due to white points detection, and on the agent decisions due to obstacles detection and ball detection.

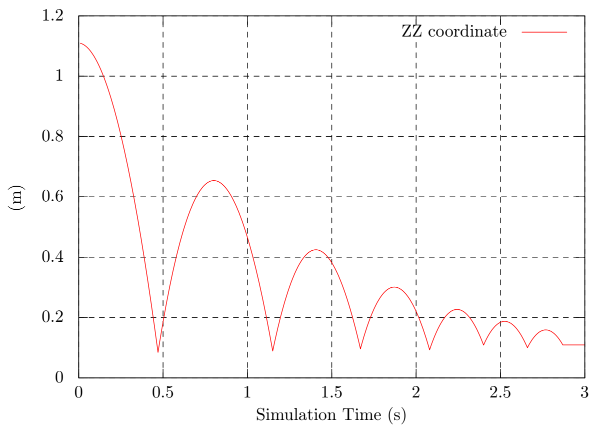
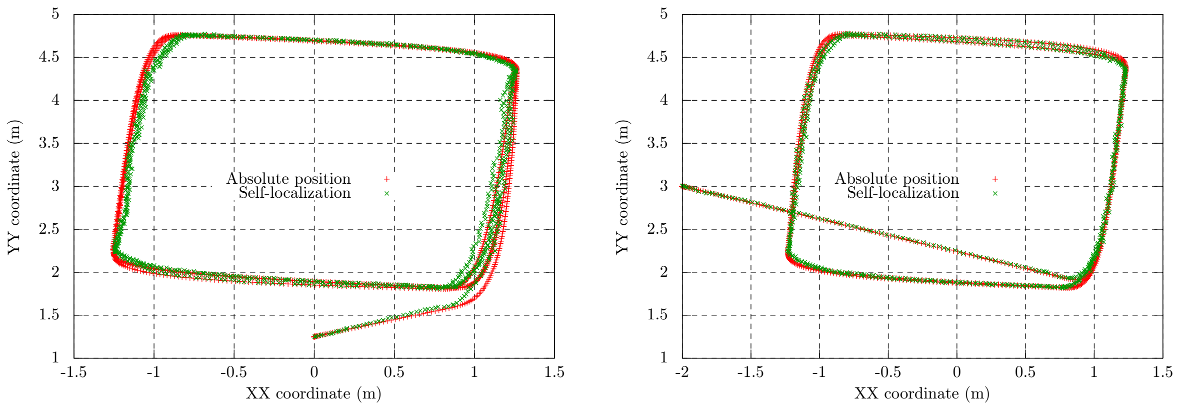


Figure 5.3: Bounciness of the ball model. Dropped vertically at one meter high, bounces six times and then stops.

White points

The virtual white points detection algorithm defaults to thirty radial sensors with three passes and it is affected by the occlusion areas created by obstacles. The results of the white points detection algorithm can be analyzed by comparing the absolute position of the robot in the simulation and the position determined by the self-localization of the agent.

The influence of the white points in self-localization is depicted in Figure 5.4. Self-localization can be achieved with just thirty radial sensors. However, raising the number of radial sensors increases the self-localization precision. The presence of obstacles can diminish the number of white points and has as consequence a less precise self-localization.



(a) With 30 radial sensors (b) With 60 radial sensor

Figure 5.4: White points influence in self-localization.

Using only radial sensors can result in missing white points, an example is shown in

Figure 5.5, where radial sensor tangential with the central circle fail to detect white points. This can be solved by using circular sensors.

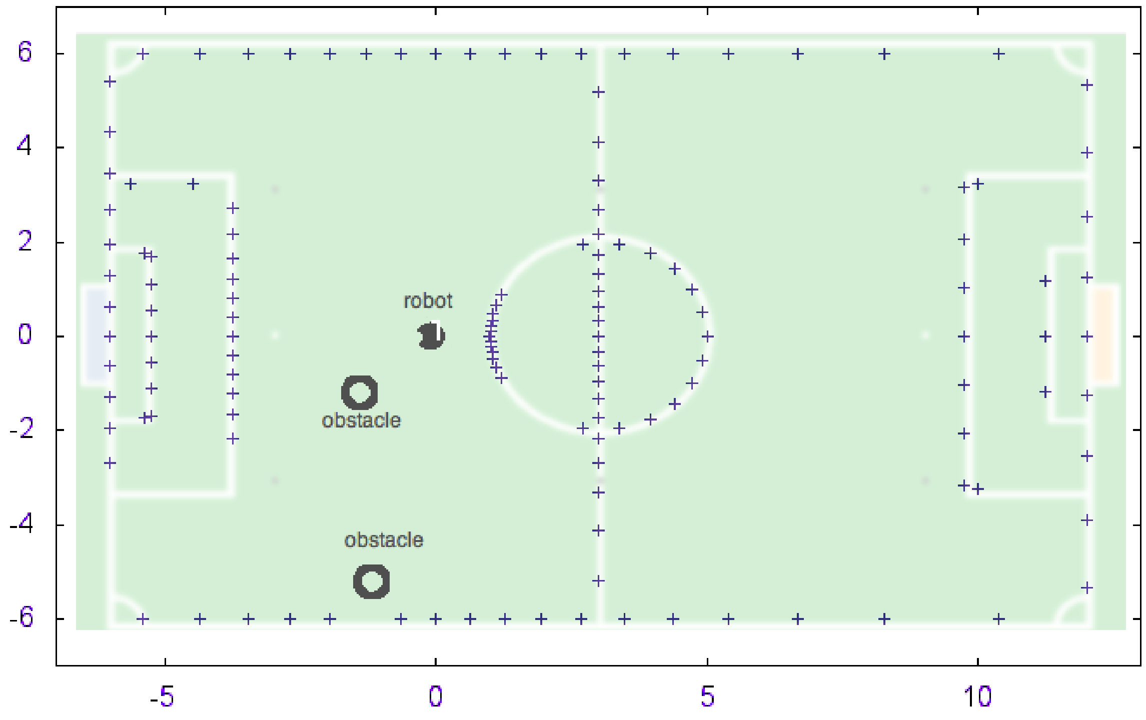


Figure 5.5: Example of a white points capture with missing white points due to using only radial sensors.

Obstacles detection

The existence of obstacles can influence the agent decisions, therefore an accurate object detection is a valuable asset specially if we are testing decision and coordination algorithms. The detection of obstacles is shown in Figure 5.6. The information generated by the virtual obstacles detection algorithm proved to be reliable for obstacles detection.

The virtual obstacle detection algorithm does not differentiate a robot model (Section 4.4.4) from an obstacle model (Section 4.4.5), they are both treated as an obstacle.

Ball detection

An example of ball detection is shown in Figure 5.7. It shows how a small variation of the ball altitude influences the detected position of the ball. When the ball is on the ground its detection is stable and reliable. Also, the ball is occluded when it falls behind an obstacle. Partial occlusion of the ball is not implemented, but due to the high dynamic position of the ball this aspect is not relevant for ball detection.

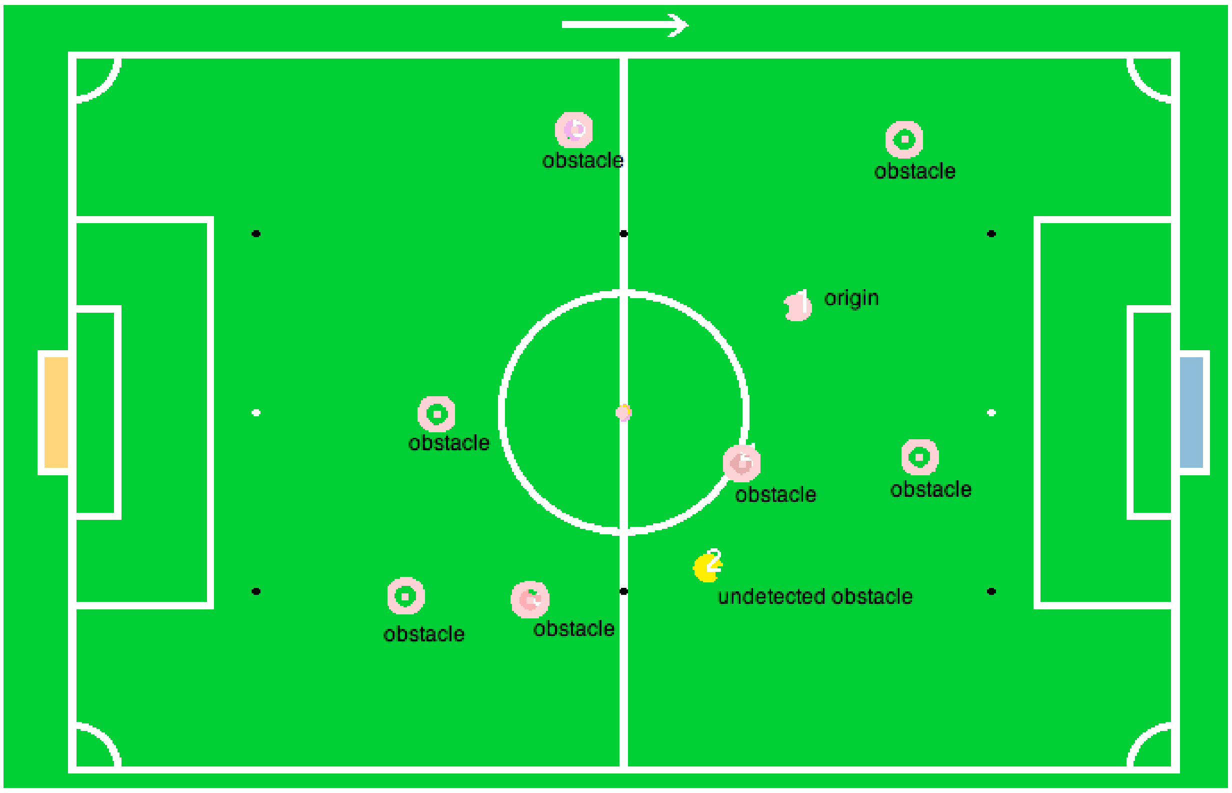


Figure 5.6: Obstacles detection with occlusion. All obstacles are relative to the "origin" robot. An obstacle is undetectable if it falls back behind another obstacle.

5.2.2 Holonomic motion

The CAMBADA team does not have the necessary resources to obtain the ground truth position of a robot, therefore to compare the holonomic motion of the real robot with the holonomic motion of the virtual robot we compared self-localization of the real robot against the absolute position and rotation of the virtual robot. To obtain the comparison data, we devised a controlled scenario that could be used in the real environment and in the simulated environment. We defined a four waypoints tour traversed by the robot while looking to a fixed point.

Using the simulator we were able to emulate the holonomic motion, shown in Figure 5.8. The holonomic motion in the simulation environment appears to slide in some turning points. Furthermore, the position over time of the virtual robot appears to be slower. Both effects can be cause by the underlining physics parameters.

The rotation of the virtual robot, depending on the situation, appears to suffer from sudden variations (Figure 5.9). Once again, the cause can be the underlining physics parameters.

The odometry of the virtual robot apparently is more accurate than the real robot odometry (Figure 5.10). The virtual robot does not have wheel encoders, and the velocity and rotation of the virtual robot frame is used to calculate the virtual robot odometry.

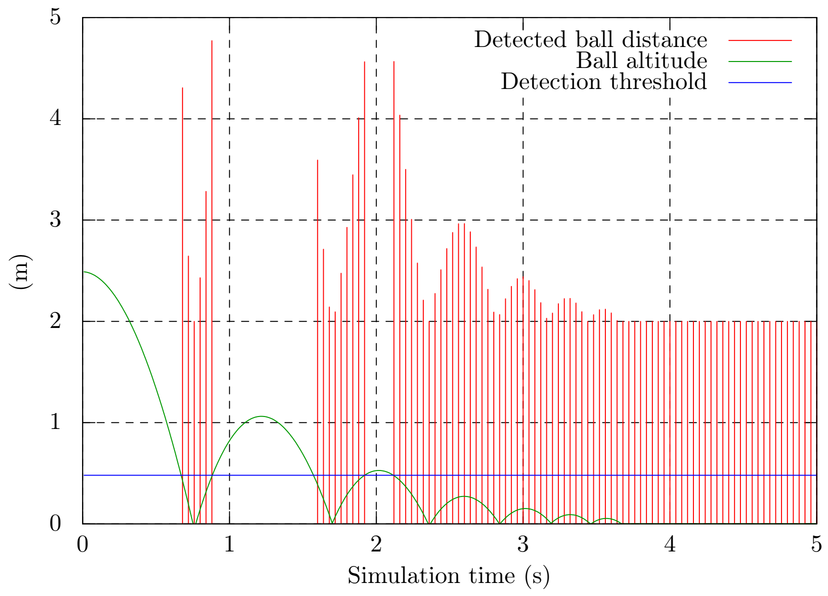


Figure 5.7: Example of ball detection. The ball projection on the ground is at a fixed distance from the robot of 2 meters. The detection threshold represent the highest altitude at which the ball is detected with a maximum *see* distance of 5 meters (Section 4.5.1).

Therefore, any external interference to the robot displacement and rotation can increase the error more than in the real robot.

ODE presented us with a problem while trying to recreate the holonomic motion in the simulation environment. After applying the correct angular velocity to the wheels, the virtual robot would only attain roughly half the linear velocity expected – angular velocity was not affected. The reason why this happens is still unknown to us. The solution to this problem was to double V_x and V_y at Equation 4.2 in the *Holonomic Motion* module.

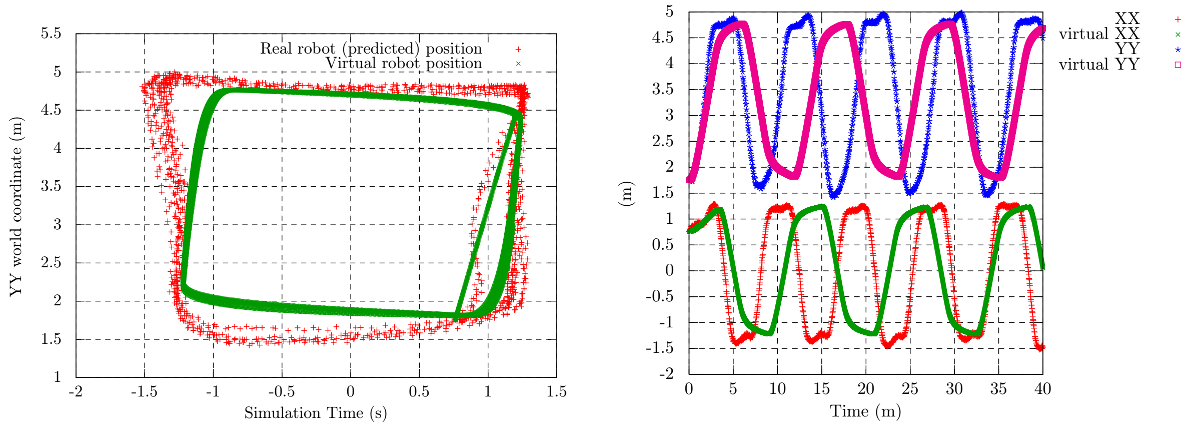
5.2.3 Grabber

The effect of the virtual grabber is shown in Figure 5.11. When the ball in range of the grabber, the applied force decreases the ball distance in a higher rate. Just like with the real grabber, the movement of the robot can result in "losing" of the ball by the grabber.

5.2.4 Barrier

The method used to know when the ball is in range is the same as the one used in the grabber. Therefore, the results obtained by the barrier are equal to to the grabber up into the *in range* detection.

The *in range* flag generated by the virtual barrier can influence the agent behavior. Setting the distance parameter with the wrong value can result in a misbehavior by the



(a) Position (b) Position over time

Figure 5.8: Holonomic motion comparison (position). Four waypoints tour while looking to the adversary goal.

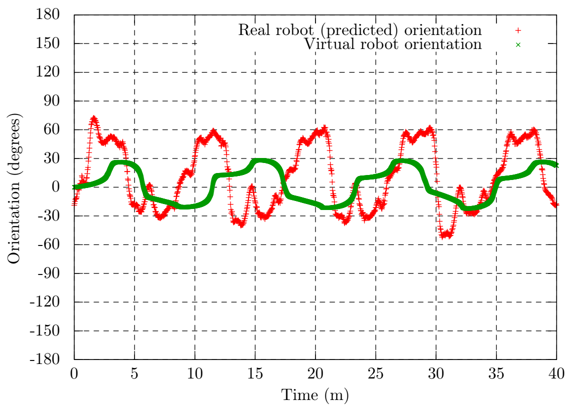
agent. If the distance is shorter than width of the inner cylinder of the robot model, the agent will never receive the information that the ball is actually is under control. As result, the agent will continuously try to bring the ball under control and will not advance to the next behavior state. If the distance is too long, the agent will assume that the is under control even though the ball is still distant. As result, the agent will be continuously losing the ball. However, up to a certain distance, the behavior of the robot combined with the grabber compensates the virtual barrier misinformation and ensures that the ball is under control.

5.2.5 Kicker system

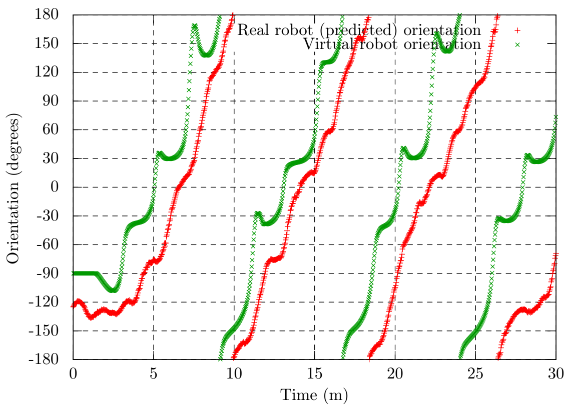
The two kick types were successfully reproduced in the simulation environment by applying a velocity given by the Equation 4.2. In fact, the data captured for the effect of the damping coefficient was reproduced by applying a high kick to the virtual ball with the virtual kicker system.

High kick

The maximum range r_m from which result the maximum initial velocity v_m applied to the virtual ball (Section 4.6.3) does not take into account the damping that the virtual ball suffers at each simulation step (Section 5.1). As consequence, in the high kick, applying v_m to the virtual ball will result in a shorter maximum range in the simulation. To obtain the desired r_m in the simulation, we incremented the r_m used by the virtual kicker system until we got the expected maximum range in the simulation.



(a) Looking to the adversary goal



(b) Looking to the ball standing in the middle of the four way points

Figure 5.9: Holonomic motion comparison (orientation). Four waypoints tour.

Low kick

The low kick is also affected by the damping applied to the ball, but we can made the assumption that the new obtained v_m will also work for the low kick. However the low kick had an unexpected problem. The low kick is essentially used to perform a pass, either for ball reposition or in-game pass. In the ball reposition the passes are usually performed with low power, i.e. low initial velocity. As result the ball was not kicked fast enough to overcome the grabber attraction force, therefore the ball would return to the *under control area*. To overcome this problem we doubled the maximum velocity v_m for the low kick.

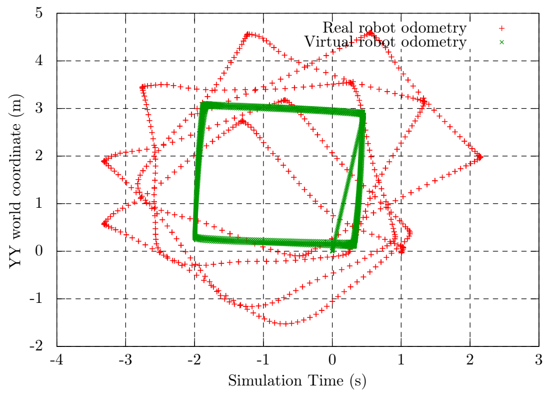
5.2.6 Compass

The information provided by the electronic compass is used to disambiguate which half of the field is which. That is done by assuming the adversary goal is at zero degrees relative to the robot which is achieved by adding the real orientation of the adversary goal to the magnetic north obtain by the electronic compass. The offset added to the electronic compass information is saved in the CAMBADA configuration file.

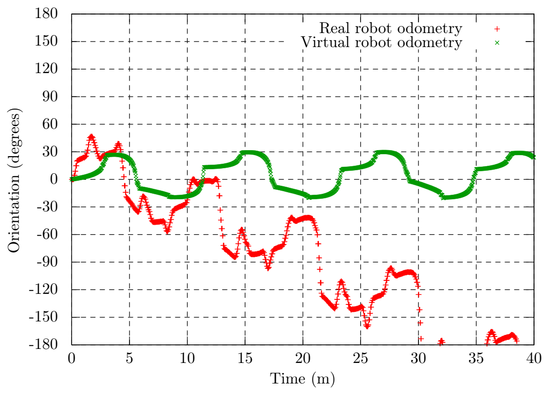
In the simulation environment the adversary goal is always at zero degrees. To avoid changing the configuration file to adjust the north offset for the simulation, the north offset is subtracted from the virtual compass orientation value.

5.3 Execution times

In this section we will discuss the overall execution time and analyze the components with relevant execution time. The reference values used in this section were obtained in a



(a) Position odometry



(b) Orientation odometry

Figure 5.10: Holonomic motion comparison (odometry). Four waypoints tour looking to the adversary goal.

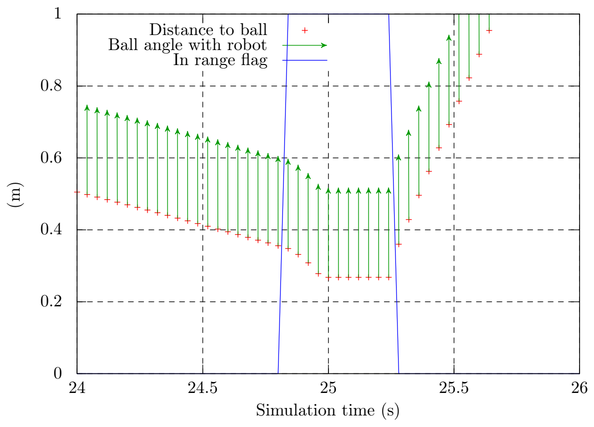
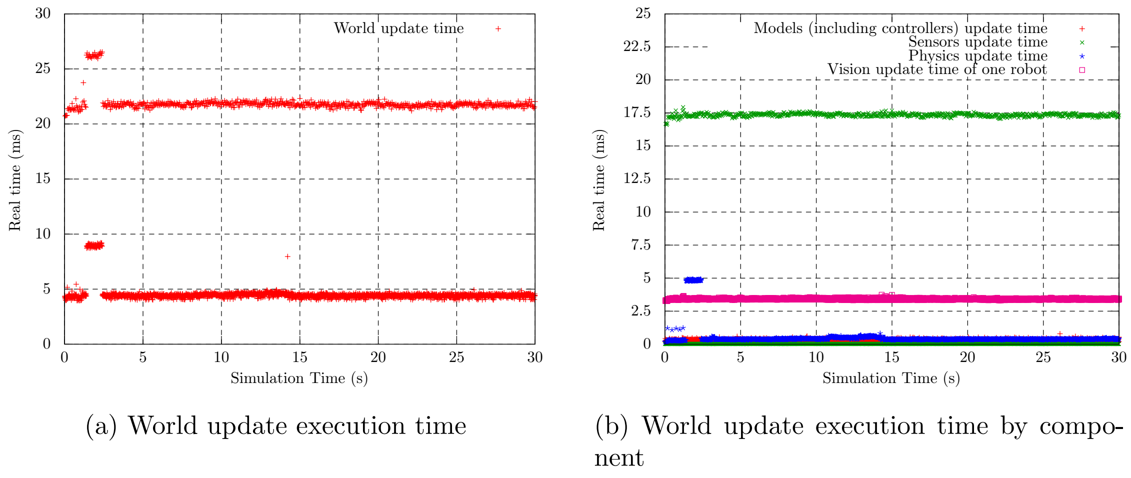


Figure 5.11: Effect of the grabber on the ball. The ball is "grabbed" and then kicked.

computer running Ubuntu 9.10 with an Intel Core 2 Duo @ 2.4 GHz and 2 GB 667 MHz SDRAM.

The world update execution time includes the bodies, geometries, controllers, sensors and physics update time. The physics step time is set to a particular value which the simulator will try to synchronize with real time after the world is updated. If the world update execution time is lower than the physics step time, the simulator will "sleep" for the remaining time. If the world update execution time is higher than the physics step time it does not cause any harm to the simulation but it will not run in real-time.

Aspects that can influence the world update execution time are the number of robots and obstacles present in the simulation, and the configuration values of the simulated sensors and controllers. We can assume that the heaviest computational simulation environment has five virtual robot and five obstacles, the world update execution time for this scenario is shown in Figure 5.12. The sudden increase of execution time at the beginning of the simulation is due to the physics execution time. For some reason, the physics execution time increases almost five milliseconds during a short period of time that only happens at the beginning of the simulation, therefore it will not be used as reference for the highest execution time.



(a) World update execution time

(b) World update execution time by component

Figure 5.12: Execution time of the heaviest simulation scenario, with physics time step of ten milliseconds.

The execution times obtained in the the worst-case scenario show us that the simulation is not capable of running in real-time at all update cycles. However, it does not cause harm to the simulation because it runs in discrete time.

The component that contributes with most of the execution time is the virtual omnidirectional vision. By analyzing the virtual omnidirectional vision execution time by operation (Figure 5.13), we found out that the obstacle detection algorithm was the sole cause for the overall high execution time in the worst case scenario. Only one virtual vision takes almost 3.5 millisecond to perform obstacles detections, considering that we have five virtual robots running at the same time it takes approximately 17.5 millisecond just to detect obstacles. The probable cause is the algorithm used to detect obstacles (Algorithm 1), optimizing this algorithm can result in better execution times.

5.4 Visualization and interaction

From the four pre-defined views offered by the GUI, the **Top view** is the most useful. Its drag-and-drop interaction with the simulation non static models provides the user a familiar form of direct manipulation. However, when the rendering widget becomes too small it becomes harder to select a model, particularly the ball model which is smaller than the robot and object models. This could be resolved either by defining a minimum size or allow the user to zoom-in and zoom-out.

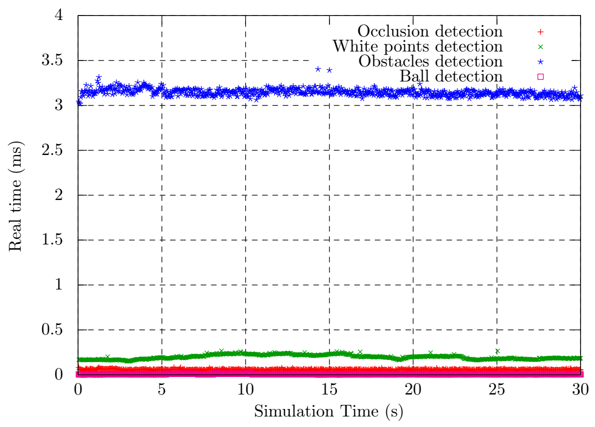


Figure 5.13: Virtual omnidirectional vision execution time in the worst-case scenario, with physics time step of ten milliseconds.

Besides the usual five virtual robots present in the simulation, the user can define the existence of obstacles models in the simulation environment in the configuration file of the simulator. Then, the obstacles can be dragged to a particular position for a specific test. By using obstacles models instead of robots models to represent an obstacle/adversary we relieved the simulator from additional workload. In addition with the capability of saving and restoring the world state the user can define a particular scenario that can be continuously repeated.

The saving and restoring world state feature is very limited. The world state is saved in memory and holds only one world state at a time. This forces the user to recreate the test scenario every time he starts the simulator and every time he wants to test a different scenario in the same simulation session. This could be resolved by adding the capability of loading and saving test scenarios from files.

When the simulation is running with the five agents at the same time, the GUI appears to sometime be unresponsive or somewhat slow, and the frame rate of the rendering engine also appears to decrease. This can be caused by the agents processes that are running on a higher and preemptive priority.

5.5 Summary

In this chapter we discussed the results obtained by the proposed simulation environment (Chapter 4). We presented the effect of damping in the ball model and how it influences the bouncing and motion of the ball model. We discussed the precision and accuracy of the virtual omnidirectional vision. We were able to present a working holonomic motion for the robot model, but without forgetting its problems and differences from the holonomic motion of the real robot. The remaining virtual sensors and controllers were also discussed. We also tried to understand how long a simulation cycle took to execute and why. We found out that it is not always possible the run the simulation in real-time. By analyzing the execution time by operation we were able to pinpoint the cause of high execution time, the obstacle detection algorithm. Finally, we discussed the developed visualization and interaction.

Chapter 6

Conclusion and future work

6.1 Conclusions

The main objective of this thesis was to create a suitable simulation environment for CAMBADA team. We started by studying the existing solutions for robotic simulation which have application on the RoboCup domain or have an strong presence in the robotic simulation world. This study helped us understand what a robotic simulation can offer, and how different, but with similar objectives, the robotic simulators are. In the end, using the requirements of the proposed simulation environment as reference frame, we compared the studied robotic simulators. Having the robotic simulators compared with each other, helped us chose one to be our code base for our robotic simulation. We chose Gazebo.

The CAMBADA robot has a wide variety of components. To be able to create a suitable simulation environment for the CAMBADA team we had to analyze all of the components and decide which one had to be simulated. Some of the non simulated components had to be modified in order to support a simulation environment, namely the RTDB and the PMan. The modifications were made with relative ease and with minimum impact to others components, meaning, the components that used the RTDB and/or the PMan did not had to be modified, or had to be modified but with small changes. The agent was the component which we were more careful to not introduce modifications, and we succeeded.

Modeling the physical entities within the simulation environment is an important task. How an entitie is modeled affect the simulation results. Besides the robot model, the physical shape of the game field, the ball and the goal were easy to model. The physical structure of the robot is more complex. We had to consider the holonomic motion system and the chassis of the robot with the cut out section to handle the ball. For simplicity, we did not model the top physical structure of the robot. Also, we had to developed virtual sensor and virtual controllers for the robot model. From the robot model we created another model, an obstacle model that can used as a simplified version of an opponent robot, i.e. without holonomic motion system, sensors and controllers).

Modeling the physics attributes of a model proved not to be a trivial matter. The ball model, is a simple model with only a sphere, however we had to considered the damping effect on the ball model that prevented it from rolling off into infinity. By applying damping to the ball model its motion is affected with relation to the damping coefficient. The damping effect on the ball model motion affects the assumptions we made while modeling the kicker system. Therefore, we had to adjust the kicker system parameters to accommodate the damping effect. The physics attributes also have heavy influence in the emulation of the holonomic motion. It took sometime until we were able to obtain an holonomic motion. However, it still needs more modeling efforts.

The developed sensors provide the agent with good sensorial information. With it the agent can successfully intercept the ball, avoid obstacles and pinpoint its position in the game field. The virtual omnidirectional vision is the Achille knee of the overall execution time, more specifically the object detection algorithm. This algorithm is very time consuming for "just" nine obstacles, and is the sole cause for non real-time simulation. It is a priority to fix this algorithm to improve overall execution time.

The developed controllers, even with some problem, were able to model physical structures without really having them. The exception is the virtual holonomic motion controller that has its physical shape modeled. The virtual robot is able to move, grab the ball and kick the ball. When performing a low kick the virtual grabber would try to bring the ball back under control, as result the ball would be kicked with a lower velocity. The solution of doubling the velocity when performing the low kick was made by simple observation which may be wrong. A solution to this problem could be modeling the physical structure of the grabber, then when a kick is performed the grabber arm would go up a little allowing the ball to leave without affecting its velocity.

The visualization and interaction component of the simulator, although not necessary for the simulation to run, is a key component. It allows the user to have a visual feedback of the changes he made to his cooperation and decisions algorithms. Furthermore, the interaction with the simulation provides the user a simple manipulation of the virtual robots and virtual ball.

In conclusion, the developed simulation environment proved to be an useful asset for the CAMBADA team. As a testing platform, the developed simulator provided several advantages to the CAMBADA team, such as save batteries power (important in competition), free the programmer from calibrations overhead, off-site programming, detect error in controlled environments, etc. At the RoboCup 2010 in Singapore, the high level software developers of the CAMBADA team, developed overnight a new ball reposition algorithm using the developed simulator away from the real robots. The motion of the virtual robot is not the same as the real robot, but that is not the most important, the most important aspect of the developed simulator is its aiding in testing and developing decisions and cooperation algorithms.

6.2 Future work

At the beginning of this thesis we stated that "Efforts are needed to create more accurate simulated models and trustworthy behaviors". Following this philosophy we should try, in the future, to evolve our models for more trustworthy behaviors.

Currently none of the sensor nor controllers simulates delays, for example, in the CAMBADA robot there is a delay between the time of perception and the time of actuation, which cannot be simulated by the developed simulator. All developed sensors are not capable of introducing noise to the gathered information, which does not allow testing in uncontrolled environments. Therefore, delays and noise should be considered in future versions of the simulator.

The model of the robot should evolve in future. Modeling the physical structure of the grabber may not be an easy task, but it can improved the kicking of the ball without having to resort to workarounds. In addition, the physical structure of the kicker system could also be modeled, completing the physical structure of the lower layer of the robot. We should also try to improve the holonomic motion, currently the the virtual robot appears to be slower than the real robot.

Fixing the virtual omnidirectional vision execution time should be a priority, namely the object detection algorithm. This would allow the simulation to run in real-time even in the worst-case scenario. With better execution we could try to create a simulation environment with two teams in the same computer, allowing in game tests. An interesting scenario would be having agents running in different computers, i.e. a distributed simulation. This rises several concerns, such as time synchronization, collisions synchronization, coherent sensorial data, and delays, but it would reduce the computational effort.

Improving the interaction with the simulation is also required. The current feature of saving and restoring the world state is very limited. Allowing the user to create different simulation scenarios, save them to a file and load them back when needed increases the productivity of the simulator. Presenting more information about the world state to the user can help debugging and analyze a test. Also, we should conduct an usability questionary which can given us a better understanding of what is good and wrong with the developed GUI.

Finally, creating a viewer that can replay a simulation can be useful for further analysis. This implies the creation of a language or protocol that describes the the simulation state as well the models present in the simulation. Additionally, it could lead to the creation of views that can connect to the simulation via networking.

Bibliography

[1] Hugo Rafael de Brito Picado. Development of behaviors for a simulated humanoid robot. Master's thesis, University of Aveiro, 2008.

[2] H.H. Lund and O. Miglino. From simulated to real robots. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 362 –365, Nagoya, Japan, May 1996.

[3] Programming - Robot Simulation. `http://www.societyofrobots.com`, last visited November 2010.

[4] Open Dynamics Engine (ODE) Community. Open Dynamics Engine (ODE) Community Wiki. `http://opende.sourceforge.net/wiki/index.php/Main_Page`, last visited November 2010.

[5] R. Hafner, S. Lange, M. Lauer, and M. Riedmiller. Brainstormers Tribots team description. In *CD procedings of Robocup Symposium*, 2006.

[6] R. Smith. Open dynamics engine. *`http://opende.sourceforge.net/`*, last visited November 2010.

[7] A.A.F. Nassiraei, Y. Kitazumi, S. Ishida, H. Toriyama, H. Ono, K. Takenaka, N. Shinpuku, M. Takaki, Y. Fukunaga, K. Yamada, et al. Hibikino-Musashi Team Description Paper.

[8] Alexander Kleiner and Thorsten Buchheim. A plugin-based architecture for simulation in the f2000 league. In *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Computer Science*, pages 434–445. Springer Berlin / Heidelberg, 2004.

[9] T. Weigel, J.S. Gutmann, M. Dietl, A. Kleiner, and B. Nebel. CS Freiburg: Coordinating robots for successful soccer playing. *Robotics and Automation, IEEE Transactions on*, 18(5):685–699, 2002.

[10] R. Lafrenz, M. Becht, T. Buchheim, P. Burger, G. Hetzel, G. Kindermann, M. Schanz, M. Schulé, and P. Levi. Cops-team description. In *RoboCup 2001: Robot Soccer World Cup V*, pages 23–61. Springer Berlin / Heidelberg, 2002.

[11] E. C. Pestel and W. T. Thomson. *Dynamics.* McGraw-Hill, New York, 1968.

[12] Simsrv, a robocup f2000 simulator. `http://kaspar.informatik.uni-freiburg.de/~simsrv/`, last visited November 2010.

[13] O. Michel. Cyberbotics Ltd. WebotsTM: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.

[14] R. Carey and G. Bell. The annotated VRML97 reference manual, 1997.

[15] Webots-User Guide. `http://www.cyberbotics.com`, October 2010.

[16] Webots Reference Manual. `http://www.cyberbotics.com`, October 2010.

[17] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[18] Gazebo, a 3D multiple robot simulator with dynamics. `http://playerstage.sourceforge.net/index.php?src=gazebo`, Last access November 2010.

[19] T.H.J. Collett, B.A. MacDonald, and B.P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005).* Citeseer, 2005.

[20] B.P. Gerkey, R.T. Vaughan, K. Støy, A. Howard, G.S. Sukhatme, and M.J. Mataric. Most valuable player: A robot device server for distributed control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, volume 12261231. Citeseer, 2001.

[21] B. Gerkey, R.T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th international conference on advanced robotics*, pages 317–323. Citeseer, 2003.

[22] R.T. Vaughan, B.P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2421 – 2427 vol.3. IEEE, 2004.

[23] B. Spitzak et al. The fast light toolkit (fltk). *FTLK: Fast light toolkit. Available: `http://www.fltk.org.`*, Last access November 2010.

[24] S. Streeting et al. Object-Oriented Graphics Rendering Engine (OGRE). *`http://www.ogre3d.org`*, Last access November 2010.

[25] Vijay Kumar. MAST simulation environment. `http://alliance.seas.upenn.edu/~kumar/wiki/index.php?n=Main.MASTSimulation`, Last access November 2010.

[26] J. Boedecker and M. Asada. SimSpark–Concepts and Application in the RoboCup 3D Soccer Simulation League.

[27] O. Obst and M. Rollmann. Spark–A Generic Simulator for Physical Multi-agent Simulations. In *Multiagent System Technologies*, pages 153–159. Springer Berlin / Heidelberg, 2004.

[28] J. Boedecker, K. Dorer, M. Rollmann, Y. Xu, F. Xue, M. Buchta, and H. Vatankhah. SimSpark User's Manual, 2010.

[29] C. Hargrove. Reflective factory. GameDev.net – all your game developing needs `http://www.gamedev.net/reference/articles/article1415.asp`, Last access November 2010.

[30] P. Riley. SPADES: a system for parallel-agent, discrete-event simulation. *AI Magazine*, 24(2):41, 2003.

[31] R. Rivest. S-expressions (draft-rivest-sexp-00. txt). *Network Working Group, available at `http://theory.lcs.mit.edu/rivest/sexp.txt`*, pages 1–11, 1997.

[32] A. Robotics. Aldebaran robotics webpage. `http://www.aldebaran-robotics.com`, Last access November 2010.

[33] C. Bustamante. Simspark monitor protocol. `http://jeap-res.ams.eng.osaka-u.ac.jp/~joschka/simspark/monitorprotocol.pdf`, Last access December 2010.

[34] P. Costa. Simtwo webpage. `http://paginas.fe.up.pt/~paco/wiki/index.php?n=Main.SimTwo`, Last access November 2010.

[35] T. Laue, K. Spiess, and T. Röfer. SimRobot–A General Physical Robot Simulator and Its Application in RoboCup. In *RoboCup 2005: Robot Soccer World Cup IX*, pages 173–183. Springer Berlin / Heidelberg, 2006.

[36] Jared Go and Browning, B. and Veloso, M. Accurate and flexible simulation for dynamic, vision-centric robots. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*, pages 1388–1389. IEEE, 2005.

[37] A. J. R. Neves, J. L. Azevedo, M. B. Cunha, N. Lau, A. Pereira, G. Corrente, F. Santos, D. Martins, N. Figueiredo, J. Silva, J. Cunha, B. Ribeiro, R. Sequeira, L. Almeida, L. S. Lopes, J. M. Rodrigues, and A. J. Pinho. Cambada'2010: Team description paper. *University of Aveiro, Tech. Rep*, RoboCup 2010.

[38] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 340–347, New York, NY, USA, 1997. ACM.

[39] H. Lund and L. Pagliarini. Robot soccer with LEGO mindstorms. In *RoboCup-98: Robot Soccer World Cup II*, pages 141–151. Springer Berlin / Heidelberg, 1999.

[40] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Systems, Man, and Cybernetics, 1999. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on*, volume 6, pages 739–743. IEEE, 2002.

[41] T. van der Zant and T. Wisspeintner. Robocup x: A proposal for a new league where robocup goes real world. In *RoboCup 2005: Robot Soccer World Cup IX*, pages 166–172. Springer Berlin / Heidelberg, 2006.

[42] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L.S. Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. In *Computer and Information Sciences - ISCIS 2004*, Lecture Notes in Computer Science, pages 876–886. Springer Berlin / Heidelberg, 2004.

[43] V. Silva, R. Marau, L. Almeida, J. Ferreira, M. Calha, P. Pedreiras, and J. Fonseca. Implementing a distributed sensing and actuation system: The CAMBADA robots case study. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 2, pages 8 pp. –788, Catania, 2005. IEEE.

[44] J.L. Azevedo, B. Cunha, and L. Almeida. Hierarchical distributed architectures for autonomous mobile robots: a case study. In *Proc. ETFA2007-12th IEEE Conference on Emerging Technologies and Factory Automation*, pages 973–980, Patras, Greece, 2007.

[45] L. Almeida, J. L. Azevedo, P. Bartolomeu, E. Brito, M. B. Cunha, J.P. Figueiredo, P. Fonseca, C. Lima, R. Marau, N. Lau, P. Pedreiras, A. Pereira, A. Pinho, F. Santos, L. Seabra Lopes, and J. Vieira. Cambada'2004: Team description paper. *University of Aveiro, Tech. Rep*, 2004.

[46] L. Almeida, P. Pedreiras, and J.A.G. Fonseca. The FTT-CAN protocol: Why and how. *Industrial Electronics, IEEE Transactions on*, 49(6):1189–1201, 2002.

[47] G. Campion, G. Bastin, and B. Dandrea-Novel. Structural properties and classification of kinematic and dynamic models of wheeled mobile robots. *Robotics and Automation, IEEE Transactions on*, 12(1):47–62, 1996.

[48] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Comput. Surv.*, 12(2):213–253, 1980.

[49] P. Pedreiras and L. Almeida. Task management for soft real-time applications based on general purpose operating systems. In *Robotic Soccer*, pages 598–607. I-Tech Education and Publishing, 2007.

[50] P. Stone and M. Veloso. Task decomposition and dynamic role assignment for realtime strategic teamwork. In *Intelligent Agents V. Agent Theories, Architectures, and Languages: 5th International Workshop, ATAL'98, Paris, France, July 1998. Proceedings*, pages 629–630. Springer, 2000.

[51] A.J.R. Neves, D.A. Martins, and A.J. Pinho. A hybrid vision system for soccer robots using radial search lines. In *Proc. of the 8th Conference on Autonomous Robot Systems and Competitions, Portuguese Robotics Open-ROBOTICA*, pages 51–55, 2008.

[52] F. Santos, L. Almeida, P. Pedreiras, L.S. Lopes, and T. Facchinetti. An Adaptive TDMA Protocol for Soft Real-Time Wireless Communication Among Mobile Computing Agents. In *Proceedings of the Workshop on Architectures for Cooperative Embedded Real-Time Systems (satellite of RTSS 2004)*, volume 2004, pages 5–8. Citeseer, 2004.

[53] F. Santos, L. Almeida, and L.S. Lopes. Self-configuration of an Adaptive TDMA wireless communication protocol for teams of mobile robots. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 1197–1204. IEEE, 2008.

[54] D. Beck, A. Ferrein, and G. Lakemeyer. A simulation environment for middle-size robots with multi-level abstraction. In *RoboCup 2007: Robot Soccer World Cup XI*, pages 136–147. Springer Berlin / Heidelberg, 2008.

[55] B. Cunha, J. Azevedo, N. Lau, and L. Almeida. Obtaining the inverse distance map from a non-svp hyperbolic catadioptric robotic vision system. In *RoboCup 2007: Robot Soccer World Cup XI*, pages 417–424. Springer Berlin / Heidelberg, 2008.

[56] T. Friedrich. *Elementary geometry.* Amer Mathematical Society, 2008.

[57] A.J.R. Neves, G.A. Corrente, and A.J. Pinho. An omnidirectional vision system for soccer robots. In *Proceedings of the aritficial intelligence 13th Portuguese conference on Progress in artificial intelligence*, pages 499–507. Springer-Verlag, 2007.

[58] P.P.R. Kit. Carnegie Mellon University, The Robotics Institute. `http://www.cs.cmu.edu/~pprk/`, Last access November 2010.

[59] J. Cunha, N. Lau, J. Rodrigues, B. Cunha, and J. Azevedo. Predictive Control for Behavior Generation of Omni-directional Robots. In *Progress in Artificial Intelligence*, pages 275–286. Springer Berlin / Heidelberg, 2009.

[60] Tom Nally. Projectile motion in 3d space. *The Liberty Basic Newsletter - Issue 130*, March 2005.