UNIVERSIDAD CARLOS III DE MADRID

Department of Systems Engineering and Automation

Bachelor's degree in Industrial Electronics
and Automation Engineering

Bachelor Thesis

IMPLEMENTATION OF A DEVELOPMENT ENVIRONMENT FOR A MINI-HUMANOID ROBOT BASED ON ROS / GAZEBO

***Author*:** Xavier Jair Díaz Ortiz

***Advisor*:** Alberto Jardón Huete

***Director*:** Félix Rodríguez Cañadillas

Leganés, Madrid
September, 2016

Acknowledgments

In the first place, I would like to express gratitude to my parents Lorena Ortiz and Javier Díaz for their unconditional love and support along my life, and for teaching me important values of effort, sacrifice and discipline to make a better person out of me, never give up and overcome all the challenges that I posed to myself. Without them, I could have never reached my personal achievements. Thank you very much!!

Secondly, I am grateful to my advisor Alberto Jardón for giving me the opportunity to become involved in a project related to robotics, which I consider now a very exciting research field where to continue my postgraduate studies, after I attended his course of Industrial Robotics in the past academic season.

I want to mention Félix Jiménez as well, my best friend since high school, who apart from being a trustworthy guy, assisted me with some good tips about how to tackle the bachelor thesis.

Last but not least, I say thanks to Patricia Muñoz and Javier Isabel for the close collaboration in the last months and in general to all the fantastic classmates and professors I was fortunate to meet at the university.

Abstract

In this project a simulation for a mini-humanoid robot, RAIDER (acronym in Spanish that stands for "Anthropomorphic Robot for Investigation and Development on Real Environments"), has been performed based on open-source robotics software tools, namely the Gazebo simulator and the ROS middleware. First, a virtual model representing the visual and dynamic robot properties has been built in URDF format to be compatible with ROS, checking its physical accuracy afterwards in Gazebo. The simulated model comprises not only the different mechanical parts of the robot and its many degrees of freedom, but also includes some of the sensors that help the real robot interact with its surroundings such as a camera, ultrasonic and infrared sensors. In addition, as Raider was originally conceived having in mind the participation in CEABOT, a Spanish competition for mini-humanoid robots, an environment corresponding to the obstacle avoidance race within this contest has been modelled partially as well.

Secondly, in order to manipulate Raider, the ros_control package along with a slightly modified version of its default plugin counterpart in Gazebo were employed. Once done, this allowed sending commands of movement to the robot joints so that links could reach goal positions with a specific desired velocity. Lastly, because apart from other powerful computing features, since the 2015a version MATLAB provides a native support interface for ROS through the Robotics System Toolbox, and because Simulink offers a more graphical and intuitive way of programming, control of all joints has been tested using this platform, which in the end has made possible the implementation of a walking gait prior to the definition of locomotion functions quite similar to the ones used on the real hardware.

Contents

Index of figures

Index of tables

Chapter 1

Introduction

In 1920 the term *robot* appeared for the first time in R.U.R (Robots Universalis Rosum), a theater play written by a Czech author, Karel Čapek [1]. The original word in the Czech language (*robota)* has actually a meaning of forced labor under slavery, which back then inspired the conception of an automated machine with human shape created to be obedient and submissive to orders given by a person. Although nowadays some of the main features that characterize a robot have gradually become true, at that time it was purely a science fiction idea still very far from real life. Later in 1942, Isaac Asimov defined in his story *Runaround* the laws that every single robot must always fulfill. These laws have had great impact on popular culture (sci-fi movies, art, print media…) and are a set of three rules stated as follows [2]:

i. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
ii. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
iii. A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

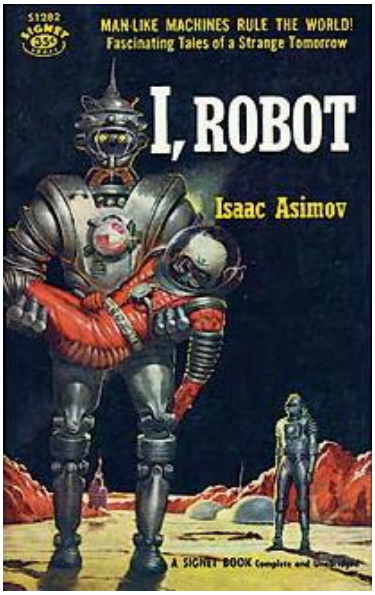


Fig. 1.1: Cover of "I, Robot" depicting the "Runaround" story where the Three Laws of Robotics were first listed

Robotics is these days a science under continuous development based on investigation, where there is always a need for innovative ideas to push the limits further in order to make robots more intelligent and autonomous. Much work and improvements still need to be done for sure, but over the last decades, robotics has experienced a strong growth as a consequence of many interesting achievements made by researchers around the world with promising results for the future. No wonder since it is a science where the most up-to-date technology is constantly applied. Different disciplines such as electronics, informatics, AI (*i.e.* artificial intelligence), computer vision, physics, mechanics and control engineering – to highlight a few – contribute to the overall progress of robotics.

Once a bit of the robotics concept and its origins have been introduced, it is now time to describe the branch of humanoid robots or also called androids, which is the subject this thesis belongs to. By definition the design of this kind of robots attempts to reproduce as much as possible or if convenient, partially, the appearance and behavior of a human being. One of the most complex aspects when controlling a humanoid and that has been the focus of many studies carried out in this field, is the coordination in real time of bipedal locomotion so that the robot can stay in equilibrium.

Currently there are several projects related to humanoid robotics, both as research done in universities at the academic level and as R&D activity at the industrial level, since many firms have started to spend resources on this branch that is foreseen to have a potential market rise in the next upcoming years. Fig. 1.2 illustrates the refinement of a fictitious robot model after each different prototype generation has surpassed the previous one. This resembles in some way the process of Darwin evolution that the *homo sapiens* has suffered in theory, though in a shorter period of time of course.

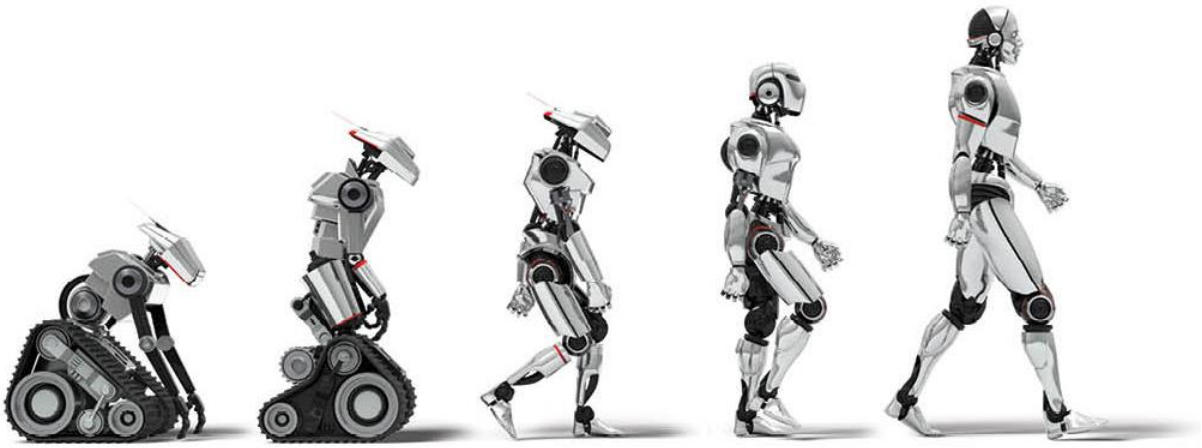


Fig. 1.2: Humanoid robot "evolution"

Within this field the most outstanding robot would be perhaps ASIMO (Advanced Step in Innovative Mobility) represented in Fig. 1.3. It was produced in 2000 by the Japanese corporation Honda and represents notably the significant dedication of the company to R&D. ASIMO is a result of the compromise made by Honda to bet on robotics in the 80's when its first bipedal robot was created. From then on, the company has invested a huge amount of money, in fact one of the largest investments in this field and for this reason, Honda has made important achievements up to the point where ASIMO is today considered to be the most advanced humanoid of the world and in general a reference for the entire industry.

The last version of ASIMO, a completely updated prototype of 130 cm and 50 kg, was revealed in November 2011. It was provided with major upgrades, for instance a new control technology of its autonomous operation, with an improved intelligence and great physical ability to adapt itself to many kinds of situations. Therefore, it is a very complex system that integrates a large amount of sensors and actuators and the control of all its components as a whole has to be done based on different previously programmed behaviors according to the stimulus received [3].

Fig. 1.3: The ASIMO robot

Another very popular, although not as sophisticated as ASIMO and significantly smaller (around 53 cm and 4.3 kg), humanoid robot found today in many academic institutions worldwide for research and education purposes is the Nao, shown in the following page in Fig 1.4. It was created by Aldebaran Robotics, a French robotics company, being first released in 2008. The popularity of this robot is so high that as of 2015, over 5000 Nao units are in use in more than 50 countries [4]. In fact, a couple of units are present at the Universidad Carlos III de Madrid.

Fig. 1.4: A group of Nao robots playing football at the RoboCup competition

After having reviewed some of the most well-known examples, it is worth mentioning that humanoids are becoming more and more capable of doing higher-level tasks, which implies an increase in the control complexity of all the subsystems built into these robots to facilitate perception and an independent interaction with the environment. This has caused in turn a drastic growth in the difficulty of the control algorithms. Moreover, as science evolves over time, the tendency is pursuing that humanoids behavior gets closer to the one of a human being. This has proven to be for the moment not so easy because a person's brain processes too much distinct information simultaneously and some areas of the brain involved in knowledge acquisition are not fully understood or even yet discovered; however, one should never underestimate the power of science and engineering in the long run.

1.1 Investigation branch of humanoid robotics at Univ. Carlos III

Within ASROB, the Robotics Association at the Universidad Carlos III de Madrid, whose logo appears next in Fig. 1.5, there is a great interest in robotics systems with morphologies of humanoid kind, existing different projects currently under development. In particular, this project has been elaborated as part of the Group of Mini-Humanoid Robotics [5].

This group was founded along the year 2006 within the Robotics Association of the university, with the aim that students could have the possibility to approach this subset of robotics during their academic training applying in a more practical way the technical skills they learn attending the several bachelor courses.

Another goal defined from the very beginning was the eventual participation in mini-humanoid robotics competitions, so that students could verify how optimal their designs were made in comparison to other alternatives developed by competitors from other universities. In this way the desire for self-improvement of the team would be encouraged, speeding up the learning process.

Fig. 1.5: Logo of ASROB, the Robotics Association of the university

1.2 General motivation of the project

With this project it is intended to create a development environment that can make simpler the definition of control algorithms utilized in mini-humanoid robots. The main features of this environment which should be taken into account are a modular design in order to allow uncomplicated replacement or reuse of the software elements involved, tracking of the trials performed and easy generation of algorithms.

In addition, a fundamental purpose of this project is that the control algorithms thought for mini-humanoid robots can be tested in the simulator before being loaded onto the real processor. Consequently, expenses resulting from damages caused by wrong implemented algorithms should be reduced. Also, students could benefit from this work in the sense that they would always be able to conduct experiments from home with a simulated robot just in case a real one is not at their disposal in the laboratory.

Chapter 2

Objectives

As already commented at the end of the previous chapter, the main goal that this project focuses on is the creation of a development environment for mini-humanoid robots, mainly characterized by an easy generation of algorithms and a flexibility of the different software blocks that will make up the simulator, so that in the future, if needed, all the required changes and improvements can be applied to the obtained ensemble, or that even the ensemble can be completely replaced. But in order to reach the ultimate goal, a series of intermediate objectives have to be attained before.

2.1 Study of the current development environments

First, the different alternatives available on the market regarding development environments have to be analyzed. Such environments should provide simulators oriented towards robotics systems and open source software will be preferred in most cases over commercial solutions. The study of these environments will help to determine which of them is the most suitable for the mini-humanoid platform.

2.2 Evaluation of convenient software to set up the environment

For building a virtual model of the mini-humanoid robot and a virtual world that tries to reproduce a certain real location where to test the robot's abilities, the use of a 3D cad program to export meshes into a simulator proves to be helpful. That's why the several options of this kind of programs will be evaluated, again favoring free license products. Apart from that, it is necessary to choose a software suite for the design of controller models, which should meet the requirements of being modular, adjustable and able to generate code in C/C++ and Arduino language or in general for some SBC (*i.e.* single board computer). In order to allow flows of data transmission between the controller and the simulator, a communications middleware that deals with this job must be chosen as well.

2.3 Creation of the development environment

Once all the components of the system have been selected, such system (the development environment) should be implemented, taking care that the conditions of flexibility and modularity are preserved to let the system be changed or improved later by other students with new suggestions and so fulfill the main goal.

2.4 Construction of a virtual model analogous to the actual robot

Whatever mini-humanoid robot is used for experimenting in real life, it is mandatory to build a virtual model of it to represent its visual and dynamic properties according to the format supported by the simulator. Besides, the various sensors that are incorporated in the robot to gather information from elements inside the simulated world must be included in this virtual model if interaction based on those sensors is wanted. On the contrary, if just test of robot movements is the only thing that matters, then simulation of sensors is not imperative at all.

2.5 Design of a controller model example to check the degree of realism

The accuracy of the built virtual robot within the simulated world should be verified in terms of physical appearance at first sight. Nevertheless, an even more relevant aspect to confirm is how realistic the robot behaves dynamically speaking. The best method to proceed with this will be the implementation of a controller model that can send commands of position and velocity to all the joints of the mini-humanoid. If no faults are detected, it would be wise to test further the coordination of more complex movements; for instance, the execution of a walking gait by the robot might be a good starting point. Finally, the chosen middleware will interconnect the controller with the simulator, ensuring a correct synchronization and functioning of the entire system.

Chapter 3

Background

In this chapter the investigation branch of mini-humanoid robots done at the Robotics Association of the UC3M - presented in the Introduction - will be explained in much more detail, since it is the framework that underlies this thesis. The mini-humanoid branch at ASROB is currently an active work still under progress. It has captured the attention of robotics enthusiastic students along the last years allowing them to become familiar with this area as well as put to practice the knowledge acquired during their academic education. Even though there is no restriction about the kind of qualification expected, the majority of these students either come from the Bachelor Degree in Industrial Electronics and Automation (which is the case of the author) or the Master in Robotics and Automation, both of them taught at the Universidad Carlos III, and in fact, many of them have written their theses to contribute in this manner with the initiative.

For the elaboration of this project the resources and facilities of the research group RoboticsLab [6], which supports the Robotics Association within the university, were used. Its logo can be seen in Fig 3.1.

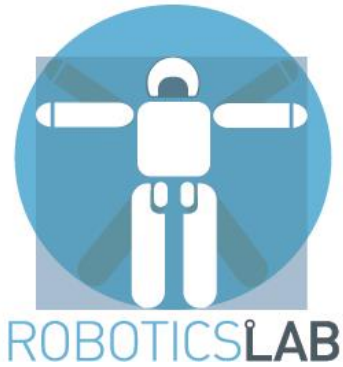


Fig. 3.1: Logo of RoboticsLab, the robotics research group from the Univ. Carlos III

3.1 Tested mini-humanoid robots

The team of mini-humanoid robots within the Robotics Association has owned in its lifetime multiple commercial kits of which primarily the Bioloid and the Robonova stand out. Such kits and its main features will be discussed in depth in the next pages, but special emphasis will be put on the RAIDER mini-humanoid platform as it is the chosen one for creating the development environment described in this document.

3.1.1 Robonova kit

From the beginning and until 2010, the platform selected for experimentation was the Robonova of the company Hitec [7], one of the most popular robots of this kind that became a success at that time. The kit comes with all the pieces in aluminum and rigid plastic material needed to build the mini-humanoid of Fig. 3.2. Once assembled, it is 30,5 cm tall and weighs 1,3 kg imitating the structure of a human being composed of a head, a trunk, two upper and two lower limbs.

Fig. 3.2: Robonova-1 kit

3.1.1.1 HSR-8498HB servomotors

In the Robonova kit there are 16 servomotors HSR-8498HB specifically designed for it (as shown in Fig. 3.3) to supply rotational motion to all the robot's joints. These servos have a turning range of 180°and can exert a torque of 7,4 kg·cm. They also expose a "Feedback Motion" feature to read from the controller the current position at any moment. Another option is the possibility to place the robot manually in an arbitrary position, query the state of all joints and save them as part of a program to merge them later into a combination of consecutive movements.

Fig. 3.3: HSR-8498HB servos

3.1.1.2 MR-C3024 Controller

The control unit of the Robonova is a MR-C3024 board illustrated in Fig. 3.4. It has a high number of ports to control up to 24 devices like servomotors. Additionally, due to its more than 40 digital inputs /outputs, a great variety of sensing elements like distance sensors, infrared sensors, gyroscopes, LCD displays, etc.… can be connected.

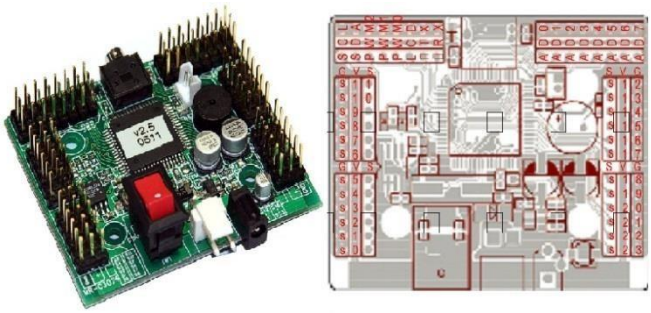


Fig. 3.4: MR-C3024 Controller

The software required to program this controller is included in the Robonova kit and is also available for free on the website of the manufacturer. It is based on the robotics programming language RoboBasic, which is a quite simple language, but very specialized and oriented to programming of robots. This language makes easier the process of programming the mini-humanoid since it contains a large amount of specific commands to control the diverse functionalities of the robot

A total of three programming environments come with the Robonova kit: RoboBasic (appears below in Fig. 3.5), RoboRemocon and RoboScript. These environments are highly intuitive and complementary to each other, but the main programming tool is RoboBasic because apart from widgets to adjust and configure the servomotors and other components of the robot, it has options to compile and load the written programs onto the controller.

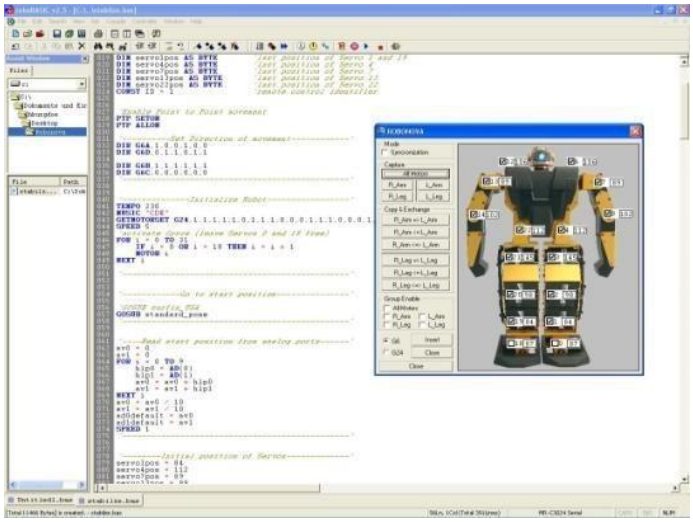


Fig. 3.5: Development environment RoboBasic for the Robonova

3.1.2 Bioloid Premium kit

In 2010 the Robotics Association bought a more modern robotic platform than the Robonova known as Bioloid [8], produced by the Korean company Robotis. In particular, the premium variant lets assemble advanced robots of up to 18 degrees of freedom. This kit is appropriate for almost every purpose ranging from just as a hobby to learning, research or competition. It contains building blocks that the user can join with screws to his liking and so the possibilities are not limited to create strictly mini humanoids, but also other types of robots with different morphologies such as animals, wheeled robots, etc. The mini-humanoid that appears in Fig 3.6 when its components are put together with that distribution, has a height of 34 cm (until the head) and a weight of 1,9 kg.

Fig. 3.6: Bioloid Premium (mini-humanoid version)

The Bioloid operation relies on an intelligent serial servo-controlled technology that provides feedback plus control of position, current, velocity, temperature and voltage of all its servomotors, called Dynamixel AX-12A. These are wired to a CM530 controller that is programmable with RoboPlus, a proprietary software of Robotis. More about these tools will be outlined next individually.

3.1.2.1 CM-530 Controller

The CM-530 board shown in Fig. 3.7 is a controller for the AX-12A servos, which in addition accepts the integration of proximity sensors AX-S1. It includes a connection to load a program from a PC that can take control over the robot. Moreover, its buttons can be used as input drives for orders or external emitter/receiver devices can be attached for control remote.

Fig. 3.7: CM-530 controller

In order to program the CM-530 the software RoboPlus is the solution delivered by Robotis. It is a free software available only for Windows computers, which implies a downside to Linux or OS X (Apple's operating system) users. RoboPlus lets the customer program each of the multiple configurations that can be built with the kit, but it is only compatible with the kits from Robotis, so it cannot be utilized along with other robotic platforms like the Robonova. RoboPlus offers the three following interfaces:

- *RoboPlus Motion*: This interface is useful to program robot movements visually. Setting a certain value to each servo (degree of freedom) the program activates the motors and synchronizes them. The motion of the robot is programmed step by step: first the robot is moved to a home position; this position gets recorded, then the robot is moved to adapt a second positon, it gets recorded again and so on. The interpolation between two positions is done automatically keeping a smooth trajectory.

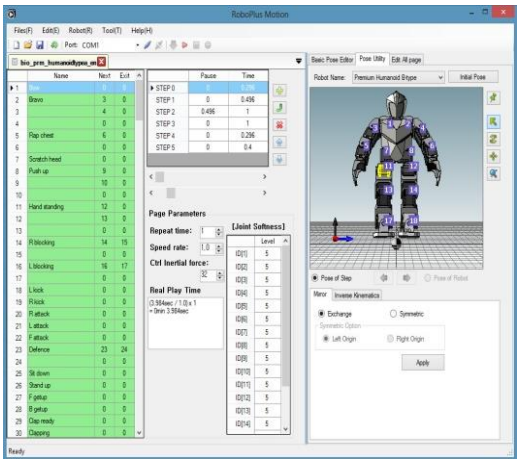


Fig. 3.8: Window of the RoboPlus Motion interface

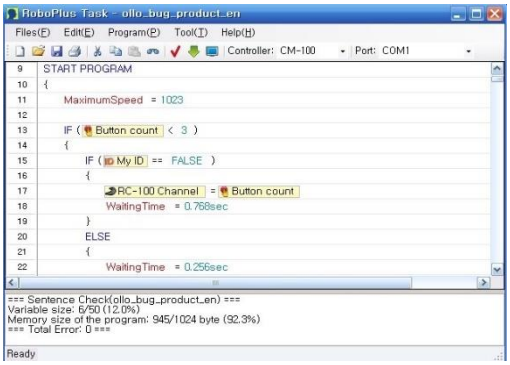


Fig. 3.9: Window of the RoboPlus Task interface

- *RoboPlus Task*: It is an editor where programming code can be written to define control algorithms of the robot's behavior, normally based on external data coming from the sensors. The response would be an actuators action.

- *RoboPlus Manager:* As its name indicates, it manages the calibration of the motors and sensors, also it serves to upgrade the firmware of the controller board. This should be done periodically to prevent a malfunction of the system. For example, ID and data transmission speed assignment for each servo is accomplished with this manager.

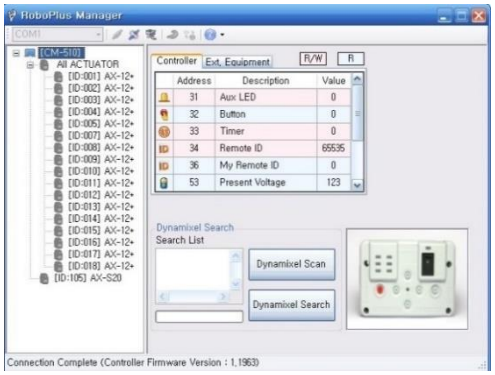


Fig. 3.10: Window of the RoboPlus Manager interface

3.1.2.2 Dynamixel AX-12A servomotors

The AX-12 is a servo produced by Robotis that plays the role of a revolute joint within the robot structure. Its position can be controlled at intervals of 0,293° thanks to its digital encoders with a resolution of 1024 bits, therefore the maximum reachable angle is 300°, if there is not a structure constraint during motion. In Fig. 3.7 the operation range of this servo is displayed, where it is clear that angles above 300° are forbidden, so it is impossible to make a complete turn of 360°.

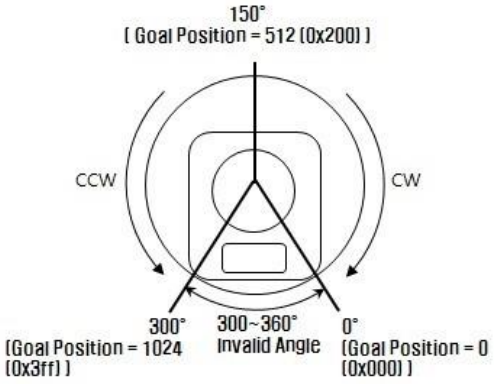


Fig. 3.11: Operation range of a Dynamixel AX-12A servo

Monitoring of these servos can give information about the load, the temperature or the current position among others. Besides, each servo has to be assigned a different ID to distinguish them uniquely when there are various connected to the CM-530 board. The user can change the IDs to match a particular order, but having in mind that only numbers between 0 and 33 are permitted. In Fig. 3.8 it can be observed how the AX-12A are connected in series to the controller through a power cable above, and a control cable in charge of transmitting data, below.

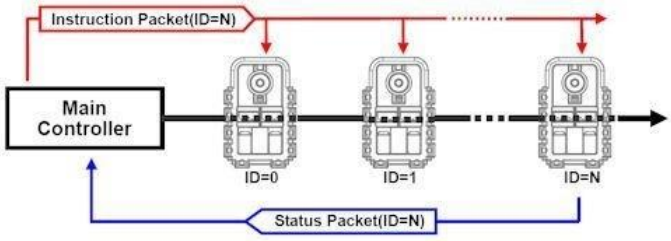


Fig. 3.12: Connection and assignment of IDs to AX-12A servos

3.1.2.3 Improvements applied to the commercial kit

Since the Bioloid Premium kit was purchased, it has undergone some modifications and improvements to expand its default capabilities in such a way that it could compete better in robotics events. For example, Fig. 3.9 shows the Bioloid by 2013 with a new head and arms. The chassis and the servos remained the same. What is more, the controller board CM-530 was replaced by the CM-900 (whose manufacturer is also

Robotis) because with the former there were limited chances for the integration of new sensors or actuators, the computational capacity was lower and it did not support high level programming languages. The latter by contrast overcame those limitations as it can be programmed with Arduino or C / C++ and offers new ports and communication buses, so the options to add more useful peripherals were extended.

Fig. 3.13: Custom Bioloid by 2013

3.2 RAIDER

Indeed, in 2014 the idea of creating the mini-humanoid platform RAIDER (acronym in Spanish that stands for "Anthropomorphic Robot for Investigation and Development on Real Environments") emerged precisely from the process of continuous improvement done on the Bioloid until then. Such idea turned into a project carried out by Javier Isabel, a student from the university enrolled in the Robotics Association. Moreover, as it is an open-source platform, the current state of his project can be checked in [9]. In the following paragraphs, the most relevant aspects that had influence on the construction of Raider will be summarized. For a detailed report, which is beyond the scope of this document, please take a look at [10] where the thesis of the author explaining his full work is referenced.

3.2.1 Selection of components

To begin with, it was necessary to pick up adequate components to equip the robot with the necessary abilities to satisfy the raised objectives. As the platform was going to be completely redesigned, in principle the components that better satisfy the requirements were going to be chosen without worrying about compatibility issues.

3.2.1.1 Robotic platform base

First, a study of the available robotic platforms was done. Since a main goal was finding a mini-humanoid robot which a vision system could be embedded into, an analysis of important properties was a must, some of them mechanical like the number and torque of the actuators and some electronic like the processing capacity and the speed of the controller. A feasible budget was also a limiting factor to be considered in the creation of Raider, since its development was financed almost entirely with the author's own funds. The requirements expected to be fulfilled were:

- Programmable in C / C++ and not dependent on any specific IDE.
- Possibility to connect a camera and write programs using the OpenCV library.
- Expandable with sensors and actuators.
- Servos of at least 12 kg·cm to support the extra load derived from the aggregation of new pieces.
- A reconfigurable and versatile chassis would have been appreciated for prospective modifications

After an initial analysis neither of the available commercial kits on the market met all the conditions to be elected. Every robot forces the buyer to use proprietary development environments and programming languages. The Robonova has too weak servos, which would have made difficult adding more weight to the robot. Therefore, the Bioloid Comprehensive kit was finally chosen because it was cheaper, easier to modify, its Dynamixel AX-12A servomotors offer a stronger torque and the package contains a larger number of them.

3.2.1.2 Selection of sensors

In order to participate in the CEABOT contest (discussed in this chapter right after the RAIDER section), and increase perception of the world it was necessary to add a list of new sensors, shown in the next figures, namely:

- Distance sensors: 2 Sharp sensors GP2Y0A21YK of type infrared and 1 sensor of type ultrasonic HC-SR04.

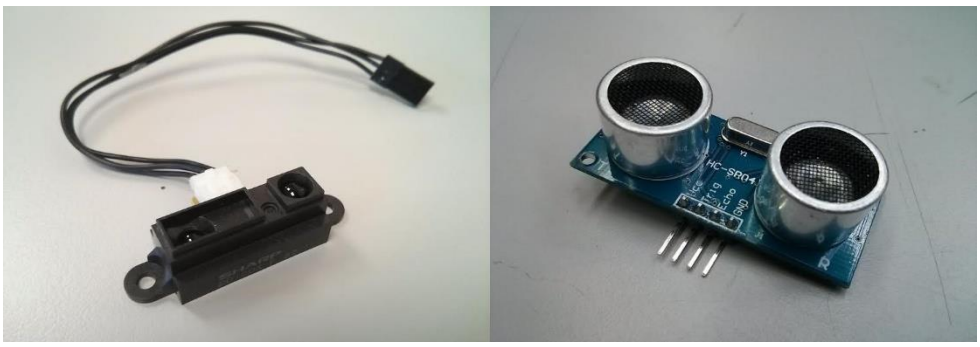


Fig. 3.14: Infrared (left) and ultrasonic (right) sensors

- Inertial sensors: two sensor types were incorporated into the robot; an accelerometer plus gyroscope MPU9150 for detection of falls and an electromagnetic compass CMPS03 for orientation

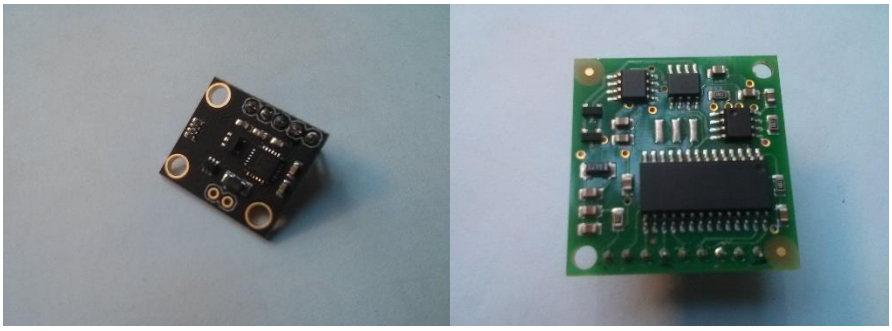


Fig. 3.15: Accelerometer plus gyroscope (left) and electromagnetic compass (right)

- Camera: a Microsoft LifeCam Cinema camera was chosen primarily because of its compatibility with the Linux driver v412 to program with OpenCV.

Fig. 3.16: Camera without the protective casing

3.2.1.3 Selection of a controller

This was a critical point to be taken into account for the correct operation of the robot. The controller CM-5 contained in the Bioloid Comprehensive package is far from ready to support the set of new devices outlined above. That's why it was substituted by a more complex system that enabled the connection and setting of the selected sensors, programming of vision algorithms and communication with the Dynamixel servos.

It was decided to adopt the solution of placing a Single Board Computer (SBC) as a main controller assigned to be responsible of the images processing task due to its great computational capacity. Alongside the SBC, a more modest controller was connected to handle the Dynamixel AX-12A servos separately.

- Locomotion controller: an OpenCM 9.04 board made by Robotis, successor of the CM900, with similar characteristics but with a smaller footprint. Its most notable specifications are a microcontroller ARM Cortex-M3, 26 input/output pins, 10 analog inputs of 10 bits and 3 serial ports (one of them reserved for the Dynamixel bus).

- Computer vision (CV) controller: a SBC called BeagleBone produced by Texas Instruments. Among its specifications there are found a processor AM335x 1GHz ARM Cortex-A8, a SDRAM memory of 512 MB, and an extensive variety of ports for the connection of different peripherals.

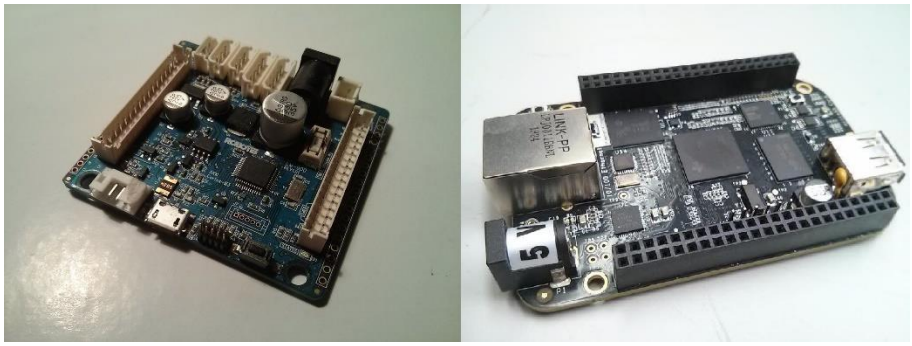


Fig. 3.17: Locomotion controller, OpenCM 9.04 (left) and CV main controller, BeagleBone Black (right)

3.2.2 Hardware architecture

The wiring diagram of all the devices that constitute the robot was designed taking care that they match their corresponding ports and power supply voltages. Such diagram is presented below in Fig. 3.18.

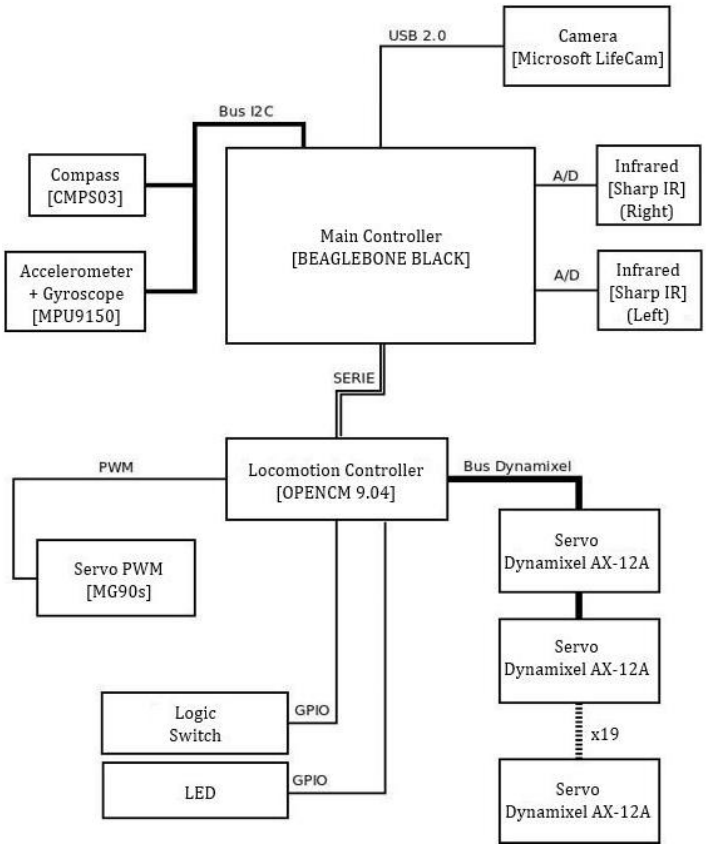


Fig. 3.18: Wiring diagram of Raider

3.2.3 Power supply unit

Due to its size and yielding, a battery LiPo Rhino of 3S and 1750mAh was selected. The dimensions of this battery and its weight (109 g, almost half the weight of the Bioloid's original battery) made it fit perfectly into the robot. After an assessment of the estimated energy consumption by all the components, it was determined that the robot would be "alive" for approximately 9 minutes without charging.

Fig. 3.19: LiPo Rhino battery

3.2.4 Structural modifications

The vast majority of the pieces that make up Raider were completely reshaped to enhance its functionalities and allow the accommodation of the new elements. For example, the forearms needed room so that the left and right infrared sensors could be inserted inside them. These pieces, parametrized to facilitate future modifications, were designed with OpenSCAD and then generated by a 3D printer. Also, apart from the default 18 servomotors that come with the Bioloid Comprehensive kit, two additional servos were added to the structure: one PWM microservo Tower Pro MG90s that acts as a neck moving up/down the camera allocated in the head, and one more Dynamixel AX-12A servo in the middle of the hip to rotate the chest, which is the same saying, to move the camera in radial direction; so both form a kind of a pan-tilt mechanism.

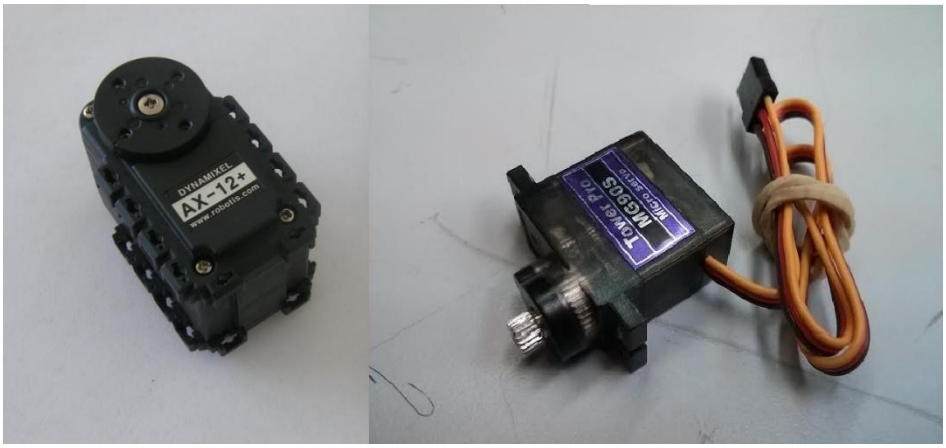


Fig. 3.20: Dynamixel AX-12A servo (left) and Tower Pro MG90s microservo (right)

Finally, once Raider was assembled altogether, its external appearance looked like portrayed in the next pictures:

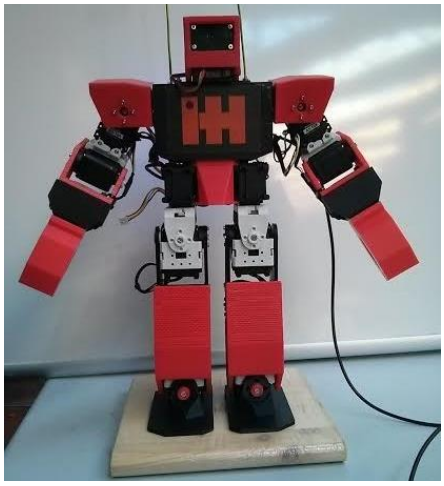


Fig. 3.21: RAIDER front view

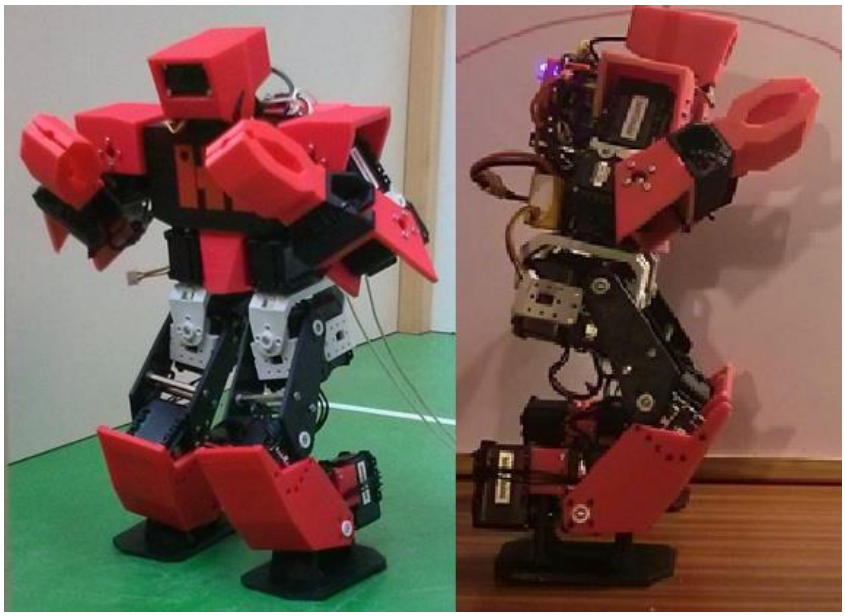


Fig. 3.22: RAIDER lateral view

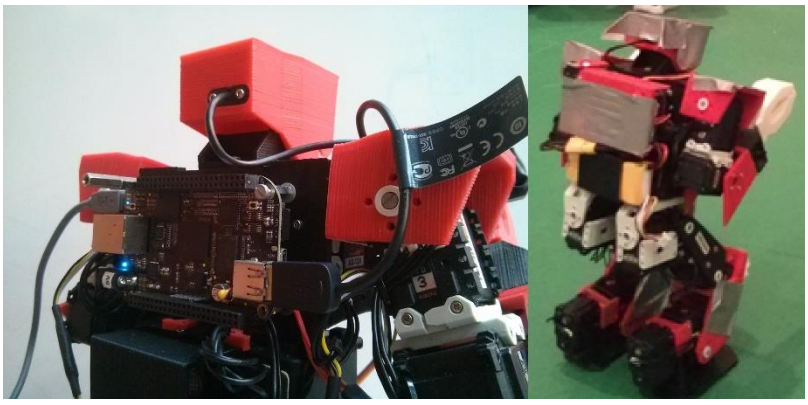


Fig. 3.23: RAIDER back view

3.2.5 Programmed behaviors

Basically, the software developed by Javier, the creator of Raider, to control the robot, was condensed in a library called *raider.h* that is the core of all the subsequent applications produced to cope with the different trials of robotics competitions like CEABOT. For instance, these include functionalities such as tracking of objects, obstacles detection and collision avoidance, finding of clear navigation pathways towards destination, getting up after a fall, interpretation of QR codes, remote teleoperation with a cell phone, etc. Some of these behaviors rely on the programming of the infrared and ultrasonic distance sensors and others are based solely on computer vision algorithms. However, from all the abilities exhibited by the real Raider, for this project only its forward bipedal locomotion will be considered to attempt mimicking a similar walking gait that shall be executed by the simulated Raider.

3.3 CEABOT competition

As pointed out in the Introduction, one of the goals that the group of mini-humanoid robots proposed itself was the participation in mini-humanoid robotics tournaments. In this context, it is worth talking about the CEABOT competition, whose logo can be seen in Fig 3.14. The involvement in this tournament promotes the learning process as well as comparing and sharing of knowledge among college students and other researchers of the humanoid robotics field.

The CEABOT competition was first celebrated in 2006 and is organized since then annually by the Spanish Committee of Automation. The team of the Univ. Carlos III has attended CEABOT from the outset obtaining positive outcomes and leaving a good impression, for instance in 2012 and 2014 the team qualified in the overall second place taking part in the contest with the Bioloid and Raider respectively.

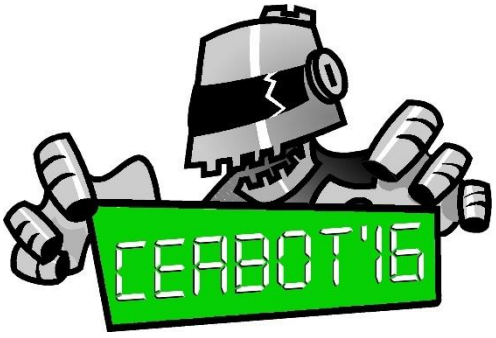


Fig. 3.24: Logo of the CEABOT 2016 tournament

Next the four events that CEABOT 2016 consists of will be explained very briefly and illustrated in the following figures. More exhaustive information about each event and the contest rules can be consulted in [11]. The elaboration of a general test environment within this project has been inspired mainly by the “obstacle avoidance race” event, but it should not be very difficult to build in the future a simulated environment of the remaining trials.

3.3.1 Event 1: Obstacle avoidance race

The robot situated in the event arena should make a round trip avoiding a set of obstacles arranged randomly by the jury. The robot departs from the middle of the start area, then it should walk until passing the finish line and finally head back to the initial point. The score will depend on the elapsed time and the number of penalties. A novelty of this year is the inclusion of QR codes on the obstacles and walls to help the robot know faster its current location.

Fig. 3.25: Obstacle avoidance race event at CEABOT

3.3.2 Event 2: Crossover stairs

The robot should reach the finish zone (only one-way) overcoming a series of 3 cm high up/downward steps. The evaluation will take into consideration the elapsed time, the number of falls and the walking straightness.

Fig. 3.26: Crossover stairs event at CEABOT

3.3.3 Event 3: Sumo wrestling

In this event a robot from one team will fight against a robot from another team inside a circular area. The battle will last three rounds of two minutes each. The winner will be the robot who has won more rounds.

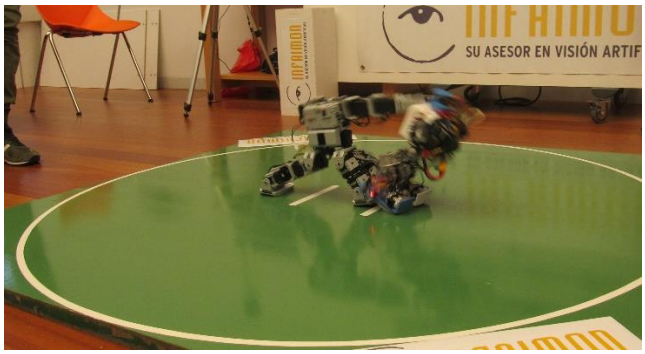


Fig. 3.27: Sumo wrestling event at CEABOT

3.3.4 Event 4: Vision test

The robot will be placed in the center of the arena facing a first obstacle with a QR code on its top portion. The robot should be able to decode this QR in order to rotate around itself and encounter the next obstacle. The bigger the quantity of recognized obstacles in the least amount of time, the better the score awarded.

Fig. 3.28: Vision test event at CEABOT

Chapter 4

State of the art

This chapter will be divided into two fundamental parts: in the first place the present state of prominent development environments dedicated to robotics simulation will be analyzed to stress the benefits of simulators: they facilitate for instance the process of designing robots and implementing control algorithms to test the operation of a virtual model before switching to the real hardware, which implies also cost savings as damages and bugs on a real robot become less frequent if a simulation has been to some extent accurately performed. Secondly, the technique of component-based modeling will be described, especially in regard to software engineering. Nevertheless, note that the software tools finally employed for the consecution of objectives within this project are not reviewed in this section, but in the next chapter.

4.1 Study of the current robotics development environments

Nowadays there is a large diversity concerning development environments that allow performing robotics simulations in a virtual tridimensional space. They offer in the majority of cases standard libraries to simulate the most common actuators and sensors that are found today in many types of robots to help them interact with their surroundings. It should be noted that the environment, also known as the robot world, where the simulation is going to take place, has to be created in any case by the user. Furthermore, apart from reproducing visually as realistic as possible the real environment, such robot worlds have to be also subjected to the different physical forces of nature like the force of gravity, friction, collision, etc.

Another important feature of simulation software to bear in mind is the support for programming languages in order to implement controllers that can manage the several functionalities exposed by the robot. Normally, languages such as C/C++, Java or Python are compatible, so if there are not any restrictions, the decision of choosing one of them really depends on personal preferences and the reached degree of mastery.

Next, some commercial simulators alternatives offered today by the market will be described remarking its main characteristics. At the market there are certainly quite a lot robotics simulation programs, thus only the ones considered to be more relevant for mini-humanoid robots will be discussed

4.1.1 Webots

Webots is a simulator developed by Cyberbotics that posseses a complete package of professional software to model, program and simulate mobile robots including of course (mini-)humanoid robots as shown in Fig 4.1. The simulator comes in three versions: PRO is the most powerful and ideal for research purposes, but its license is also the most expensive; EDU is not so powerful, but it is cheaper and good enough for robotics education; MOD is free, however it lacks many features and in order to run a simulation, robots must be purchased separately, so new robots cannot be modelled in this version. With Webots, the user is able to design both the different parts of the system that the robot comprises and the virtual world in 3D space. Also many appearance object properties like the shape, color, texture, and physical like mass, inertias, friction can be adjusted to match real life expectations.

In addition to the mentioned characteristics, various options to equip the robots with useful devices are present, since it contains libraries that let simulate a wide range of sensors and actuators such as cameras, servomotors, transmission mechanisms, distances sensors, contact and force sensors, receiver /emitter of radiofrequency signals among other items of a very long list. The fact that Webots is used by over 1261 universities and research centers worldwide reflects its high popularity [12].

In regard to the controllers that can be made with Webots, these can be written inside the text editor of the simulator itself, and externally from another IDE that supports languages like C/C++, Java, etc. Once the control has been implemented, it is possible to test the behavior of the robot governed by the controller in a certain world with realistic physical laws, so that later the code can be transferred to the robot's real processor.

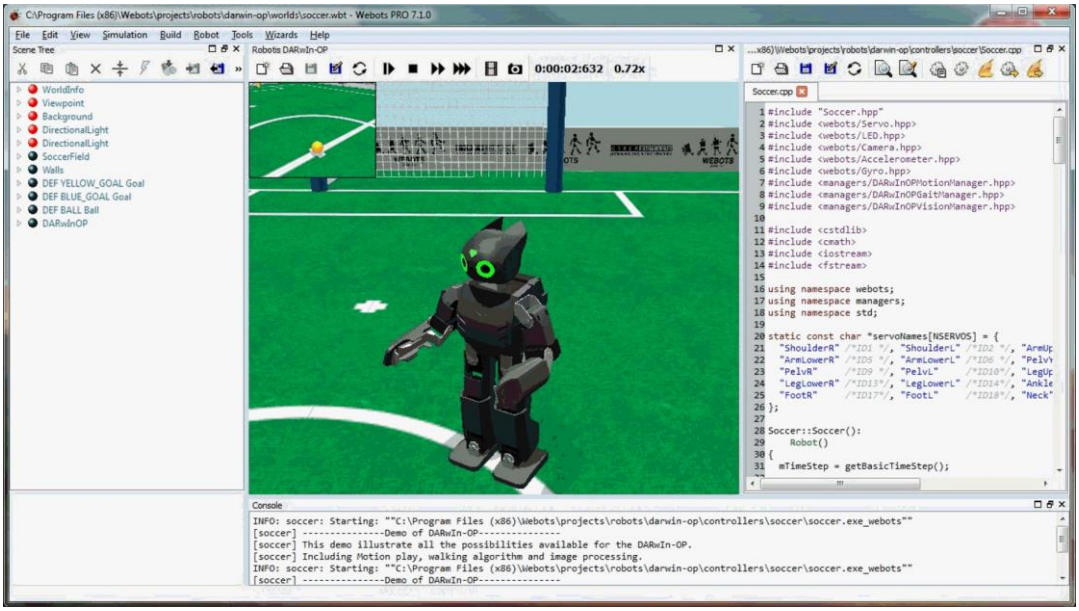


Fig. 4.1: Webots development environment

4.1.2 V-REP

V-rep (Virtual Robot Experimentation Platform) is a simulator made by Coppelia Robotics whose slogan says “Create. Compose. Simulate. Any Robot”. It is very versatile and optimal for several kinds of applications like fast algorithm development, remote monitoring, safety double checking, fast prototyping and verification, factory automation simulations, and robotics related education.

As long as it is not used for commercial work, the version PRO EDU is free and open source. It is available for Windows, Linux and OS X, so this cross platform characteristic allows that the contents created can be portable, scalable and easy to maintain. One of the main characteristics of this simulator is its distributed control architecture, which means that anything – from a simple object, sensor or actuator to whole robotics systems - can be controlled separately via six programming approaches: an embedded script, a plugin, add-ons, a middleware node, a remote API client, or a custom solution.

In order to simulate the interactions between objects and the many aspects that characterize a robot, V-rep provides useful tools such as the possibility to try four different physics engines, inverse/forward kinematics solvers for any kind of mechanism, collision detection between meshes or point clouds, minimum distance calculation, a path/motion planning library, powerful proximity and vision sensors simulation, etc. On the other hand, controllers can be implemented thanks to powerful APIs in any of the seven supported languages, namely C/C++, Python, Java, Matlab, Lua, Octave and Orbi [13].

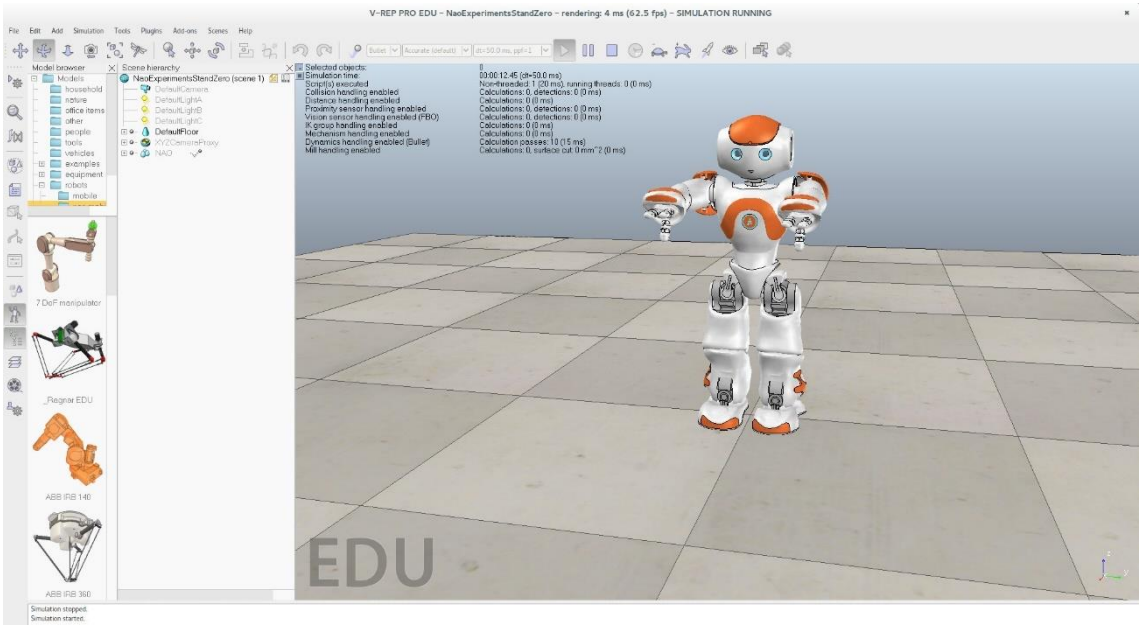


Fig. 4.2: V-REP development environment

4.1.3 OpenHRP3

OpenHRP3 (Open architecture – centered Humanoid Robotics Platform version 3) is an open source simulator specialized for humanoid robots, that allows the user examine and modify the original model of the working robot. Furthermore, the control loop can be programmed and tested through a dynamic simulation. In conjunction with OpenHRP3, a series of calculus libraries are installed to facilitate the elaboration of robotics software.

This simulator is framed in a plan of development and investigation sponsored by the Japanese Ministry of Economy and Industry; additionally, it is part of the project "Distributed component type robot simulator" driven by the association "Cooperation of Next Generation Robots". Also organizations like the University of Tokyo and the Japanese "National Institute of Advanced Industrial Science and Technology (AIST)" cooperate to the progress of OpenHRP3.

The models of the robot and its world are stored and distributed as VRML format. Within the simulator there exists functions for the implementation of control drivers, which support the dynamics of diverse mechanisms, and visualizations functions to monitor the simulation process. The two fundamental modules for the user are the "MotionPlanner" with which avoiding collisions trajectories can be spawned, and the "PatternGenerator" that serves to produce stable robot locomotion movements based on the control of the ZMP (zero moment point).

The main drawback of this simulator is that its realization has revolved too much around its own robot, the HRP3, therefore its adaptation to other different humanoid platforms is not so straightforward. Fig. 4.3 illustrates down below the graphical interface of the development environment where precisely the HRP3 is being simulated [14].

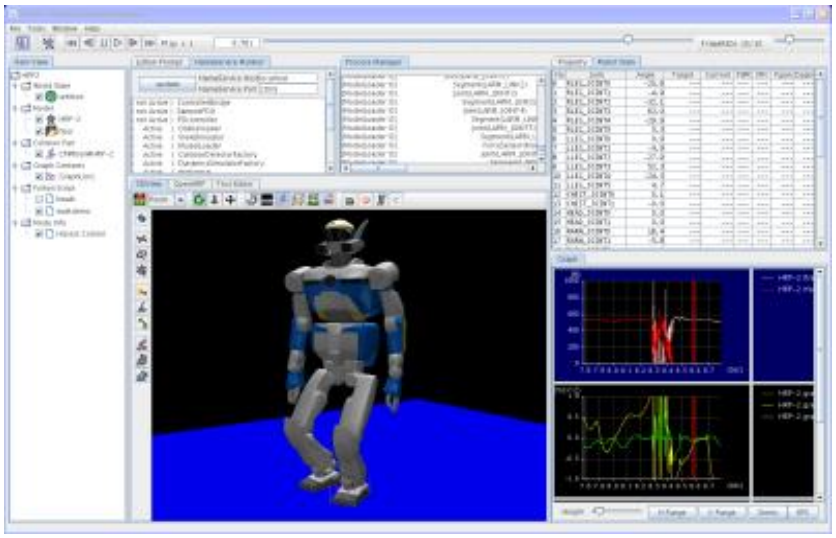


Fig. 4.3: OpenHRP3 development environment

4.1.4 SimRobot

The main characteristic of this simulator is that it was defined to support all types of mobile robots among which mini-humanoid robots are embraced. The XML language is used to build all models inside the simulation program, so it is likely to specify any type of robot and also the environment without needing to make use of another complementary programming language.

Within the models written in XML, all components of the robot body can be included as well as multiple sensors and actuators. In this manner, the user has considerable freedom when modelling his prototypes. While the physics simulation of rigid bodies is done in SimRobot by ODE (Open Dynamics Engine), the graphics are rendered by OpenGL. In Fig. 4.4 the Robonova is displayed in this simulator.

Both the University of Bremen and the German Research Center of Artificial Intelligence contributed to the development of SimRobot around 2004, initially targeted to help the German team replicate the circumstances of the RoboCup competition. It is being used now in some European universities for further research on autonomous robots [15].

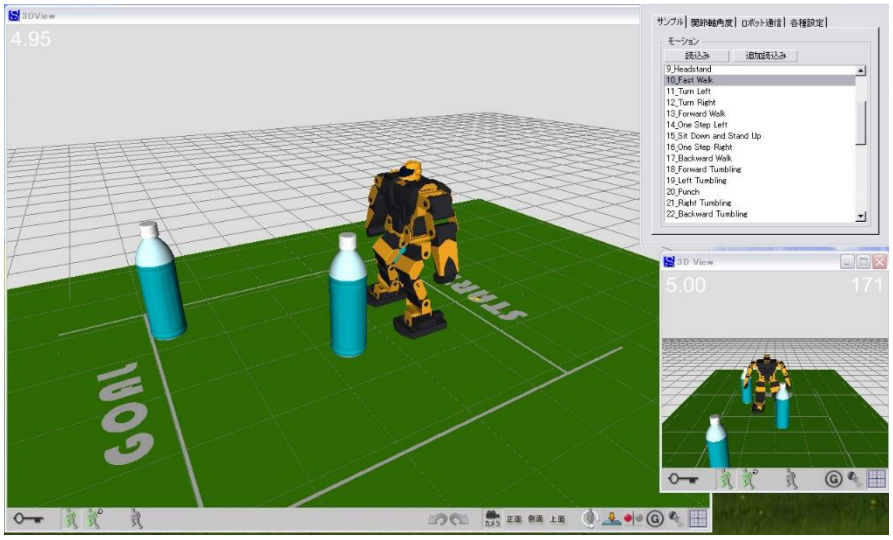


Fig. 4.4: SimRobot development environment

4.1.5 OpenRAVE

OpenRAVE is an open source simulator that provides an environment for testing, developing, and deploying motion planning algorithms in real-world robotics applications. It was created by Rosen Diankov as part of his PhD thesis in 2006 at the "Robotics Institute" of the American "Carnegie Mellon University" Its major focus is on simulation and analysis of kinematic and geometric data related to motion planning.

The independent nature of OpenRAVE permits an uncomplicated integration into existing robotics systems. It offers many command line tools to work with robots and planners, and the run-time core is small enough to be used inside controllers and bigger frameworks. An important target application is industrial robotics automation. Fig. 4.5 shows the GUI, where the Romeo humanoid robot is being simulated.

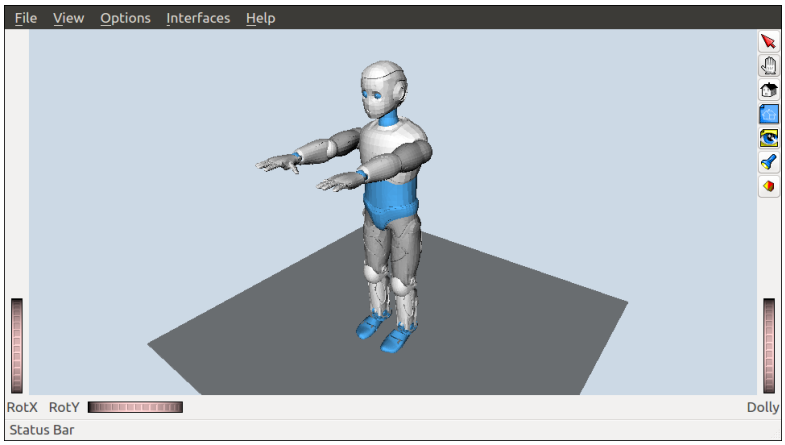


Fig. 4.5: OpenRAVE development environment

4.2 Component-based modeling

This kind of design is a trendy technique in most industrial sectors, but particularly in the field of software engineering due to the advantages it brings in comparison to other modeling design approaches, even so, in certain areas it may not be applicable when the implicit cons may exceed the pros.

The complexity and size of applications increment fast and concurrently high-quality software products are being demanded, which involves a cumbersome development and always in the shortest time possible. Consequently, in order to be able to fulfill the expectation, it is indispensable to develop applications that can be recycled for an eventual integration into a new system and this is exactly the objective that component-based modeling tries to address.

Throughout this section such modeling technique will be commented, starting with the definition of what a component means in this situation, then covering the development methodology and the application design, the fundamental attributes that should be satisfied by a component-based model; and finally the concept of robotics middleware will be introduced as a perfect example of software that enables creating an infrastructure for robots made of several standalone components.

4.2.1 Fundamentals of component-based modeling

The professionals of software engineering are currently inspiring themselves by the way other engineering fields have resolved their problems for over the last century; as Chambers [16] predicted, it was becoming a necessity to import to this domain the concept of component as a building block. Software components are designed in such a way that they can be reused in a variety of applications without having to be modified, allowing the creation of new components and applications from already existing ones.

Nevertheless, software design continues to be a complex activity, since it is not possible that all new applications can be built merely on the basis of preexisting components. The nature of software is unequal from other scientific areas, because it has to keep up with hardware restrictions, so applications optimized for some systems, may not be optimal for others.

4.2.2 Concept of component

A software component could be defined as a binary unit of software applications that has a set of interfaces and requirements, which can be developed, acquired, incorporated to a new system and combinable with other components independently in time or space.

This concept should be analyzed from three different perspectives to get a complete understanding of its dimension. These three perspectives were proposed by Brown [17] to be able to compare differing based-component models:

- Packaging perspective: the component is treated as a packaging and distribution unit that can be organized and recognized with a series of elements for later reuse in the same or in a new project.
- Service perspective: the component is considered as an entity that advertises a service to potential clients. The services are grouped in interfaces specifying how the requests and responses between actors should be carried out.
- Integrity perspective: from this point of view the borders and limitations of each component are identified to determine how to proceed correctly in case of a future substitution.

Apart from the consideration of these perspectives, it is important to anticipate the size of the components. As each of them can be viewed as a replacement unit, their granularity has to be adjusted to the purpose and the role played by the total application. Also their functionality should be well devised; there are for example graphical interface components, input/output components, control components, synchronization components, data analysis components, storage components, etc.

4.2.3 Stages of component-based modeling of systems

The development of systems in the software engineering framework has followed traditionally a top-down approach, but component-based modeling suggests the opposite, a bottom-up strategy to implement final products through the assembly and integration of preexisting software elements. Basically, a design based on components can be done in two ways: select and aggregate components previously delivered by third party suppliers, or develop oneself the modular components that will form the particular application. Depending on the decision taken, the development cycle might be slightly different, but it will look anyway pretty much like the one depicted in Fig.4.6, defined as COTS (Commercial-off-the-shelf) by Carney [18] and whose guidelines are:

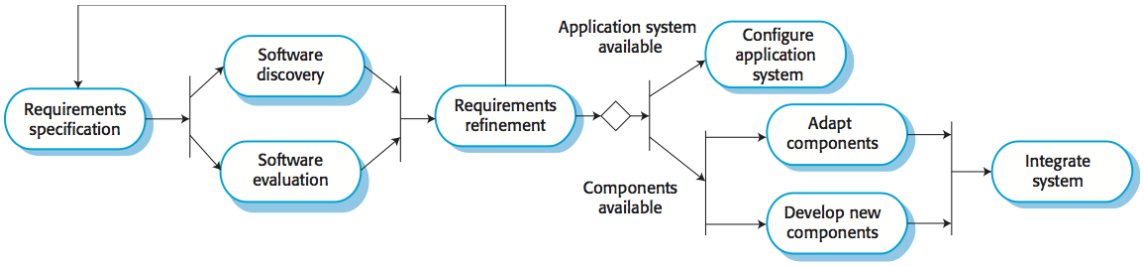


Fig. 4.6: Stages of component-based modeling

4.2.4 Example: Robotics middleware

Robots are complex systems that are composed of many non-homogeneous hardware and software elements. The proper coordination and interaction of these components needs to be handled in a way that for example, together they allow the robot achieve a common goal and overall operate successfully as expected. Therefore, in this sense a middleware turns out to be an essential instrument in order to manage the complexity and heterogeneity of robotics systems. The term *middleware* comes actually from the field of computer science; Bakken et al. [19] stated the definition of middleware as follows: "… a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system". Fig 4.7 illustrates that a middleware is indeed an abstraction layer that lies "in the middle" – hence its name – between the operating system and software applications on each side of a distributed computing system in a network; so that's why it is also sometimes referred as an almost invisible "glue code" that should enable the coupling of different subsystems without adding too much overload or extra restrictions on the components, though in reality compromises have to be made.

The boundaries that distinguish a middleware from the operating system are up to a certain point not very precise (in the past TCP/IP stacks were provided by a middleware, but now they are integrated into every OS), however typical examples are web servers, database services, content management systems, messaging-and-queueing software, transaction monitors and telecommunications software.

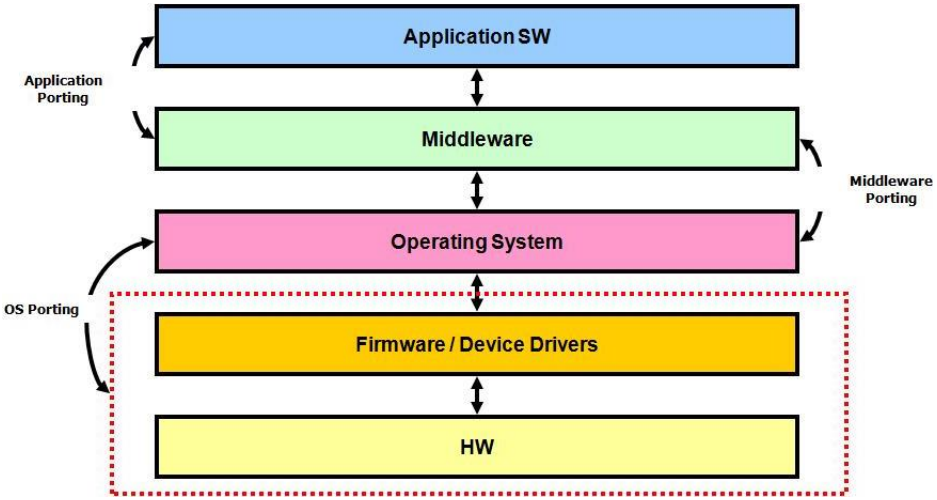


Fig. 4.7: Middleware location inside a layered system architecture

Generally speaking, there are many pieces of software running on a robot responsible for accomplishing different tasks like reading sensors data, passing and processing information coming from different sources, executing instructions to drive the actuators, etc. The design of a custom solution consisting of a single application that manages all these functionalities appears to be tedious and at the same time impractical for code enhancement and sharing of projects. By contrast, a robotics middleware is flexible, robust, scalable and upgradable as the size of components grow, which results profitable for developers because it provides them with more convenient, standardized hardware APIs that favor portability, reliability, software modularity and the necessary abstraction of hardware to hide the low-level -specific details of devices. In this way only the control algorithm has to be implemented as a component, and later it can be combined and integrated with other existing components. Moreover, if an outdated component has to be improved, modified or replaced by a newer version, a middleware allows this to be done without altering the rest of the system, causing an increase in efficiency of code reuse.

A middleware can also establish a communication channel where data is transmitted from one component to another, as long as the interfaces between them are kept coherent. Thus, in such scenario, the mentioned advantages of a robotics middleware lead to a better organization and maintainability of the project, followed by a reduction of development costs and programming efforts.

Chapter 5

Review of deployed software

In this chapter the several and most significant software tools that were used to create the components of the development environment for the Raider mini-humanoid robot will be described remarking the most relevant characteristics for this project.

5.1 ROS

ROS (Robot Operating System) is among other things an open-source robotics middleware, but it presents quite a lot of useful software that make it really much more than just that. The official definition of ROS is "… a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" [20]. ROS is a huge project that has received along its history the contribution of many people and institutions, but it was initially developed in 2007 by Willow Garage, at that moment a robotics start-up company, and recently in 2013 the Open Source Robotics Foundation (OSRF) assumed the responsibility for its maintenance and core development until today.

Like most open-source projects, it was built to promote cooperative work between robotics research teams specialized in different areas. As a result, the ROS community has experienced a strong growth over the years (has even a dedicated wiki and Q&A website to share knowledge) and multiple versions have been released. The current version is ROS Kinetic, however for this project the so called ROS Indigo version, which runs on a computer with Ubuntu 14.04 installed, has been preferred due to a better compatibility with the tutorials available online. Its logo is shown below in Fig 5.1

Fig. 5.1: Logo of ROS Indigo Igloo

5.1.1 ROS features

ROS software is distributed in the form of packages that encapsulate executable and source files for a concrete purpose. Some of the main functionalities it provides are:

- Communications middleware: this is the most important feature for this project along with the use of the ros_control package. Basically it offers a message passing infrastructure for inter-process exchange of information based on a publish/subscribe mechanism with the possibility to record those messages as "rosbags" and playback them later in the network for reproducing the same conditions. But because publish/subscribe is not done simultaneously, ROS also offers a synchronous interaction between processes similar to a client-server relationship. This will be commented more in depth in the next heading. Also it offers a distributed parameter system so that different tasks can share configuration settings, kind of like global variables work in other programming languages.

- Robot specific features: these include standardized robot messages that cover most robotics use cases (for instance sensor, odometry or pose data); a robot geometry library to handle coordinate transforms between different frames, a robot description language known as URDF (Unified Robot Description Format) to express the robot's physical properties, sensors and appearance; diagnostic means to check the correct state of the robot or detect unexpected problems; out-of-the box pose estimation, localization and navigation capabilities, etc.

- Other tools: Rviz, a graphical interface to monitor many sensor data types (laser scans, camera images, point clouds) and visualize in 3D space the URDF-described robot and its many coordinate frames (no physics engine involved though). Besides, there is rqt, a tool based on Qt to configure custom GUIs that can display valuable information about the robot or plot graphs of data over time. Moreover, for users that feel perfectly comfortable working only with the terminal, the ROS core functionalities and introspection utilities can be accessed completely through more than 45 commands line tools.

- Integration with other libraries: if the above mentioned characteristics are not enough, ROS can be integrated further with other interesting platforms such as robotics simulators, OpenCV and PCL (the Point Cloud Library) for powerful computer vision algorithms and image processing; the "MoveIt!" motion planning library and ROS-Industrial, an extension of the default ROS core that is oriented especially towards manufacturing automation and robotics.

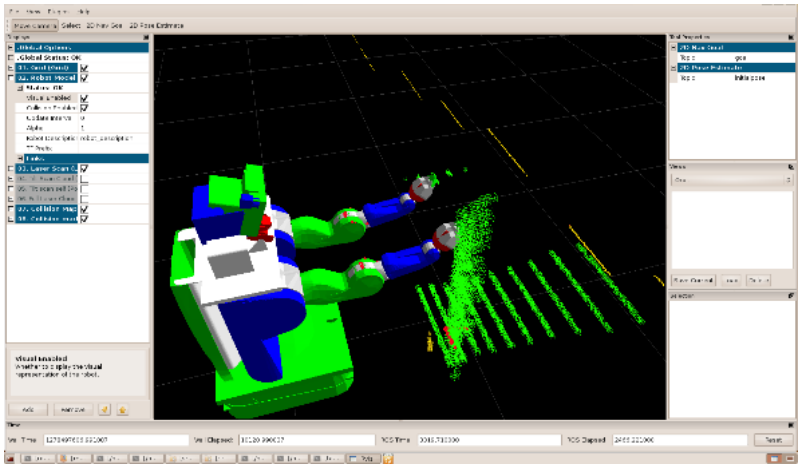


Fig. 5.2: Rviz, visualization tool of ROS, displaying the PR2 robot

5.1.2 ROS computation graph level

The concept *Computation Graph* of Fig 5.3 refers in ROS to the peer-to-peer modular network of processes that are running, separately designed and then loosely coupled at runtime. In ROS terminology the components of this network are known as follows:

- Nodes: are processes responsible for computing some specifically task like reading sensor data, robot state monitoring, control of motors, etc.
- Master: provides name registration and lookup to other members (*i.e*. nodes) inside the network.
- Parameter server: it is a part of the Master where nodes can store data by key and retrieve it for their configuration.
- Messages: are used by nodes to communicate with each other. They are data structures that have type fields, which in turn can be standard primitive data types (boolean, integer, float…).
- Topics: are content identifiers of a message. A certain node that sends out a message to a topic is called a publisher and a node that "listens" to a topic is called a subscriber.
- Services: complement the many-to-many and only one-way data transmission of the publish/subscribe mechanism when there is a need for a request/reply relation. A node sends a message to another node that advertises a service and waits for the reply, much like a client-server interaction.
- Bags: are a format in which messages can be stored to be populated again later in the network. They are usually helpful to test algorithms.

A node registers its name and service/s with the ROS Master and receives in return the contact information registered by the rest of the nodes, so a direct connection between nodes is established as the Master only provides lookup information. Therefore, a subscribing node will request to connect to the node that publishes to the topic of its interest. Multiple nodes can communicate each other at the same time agnostically; that means they don't care which nodes are playing or not the role of publisher /subscriber they simply send/listen messages to/from a topic. Internally ROS makes use of standard TCP/IP sockets for its TCPROS protocol [21].

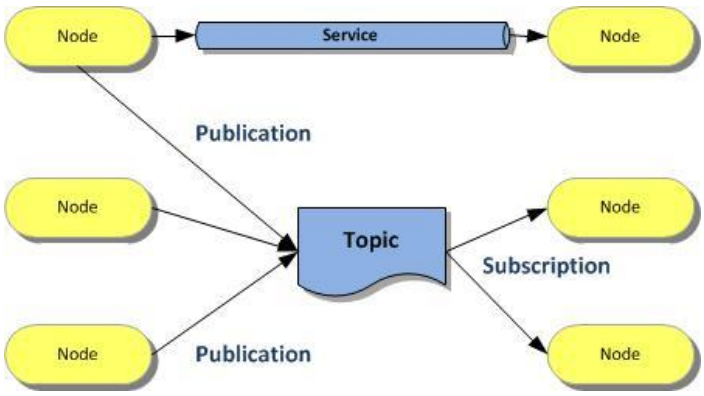


Fig. 5.3: ROS computation graph

5.1.3 ros_control

The ros_control package [22] is a key ingredient in the elaboration of this project. As its name suggests, it provides a series of useful tools to facilitate controlling a ROS-enabled robot. An overview scheme can be seen in Fig. 5.4. Essentially what it does is creating an abstraction of the robot hardware to hide the low-level details of the devices, so that controllers don't have to interact directly with the real or simulated HW (hardware). This abstraction appears on the right of Fig 5.4 as RobotHW, which is nothing more than a base class; where the robot joints, considered to be resources, are contained. A group of resources that share the same control method (position, velocity or effort), is treated in turn as a read-write hardware interface and all of them expose also read-only JointStateInterface, which gives information about the current state of the joints, so a robot is in ultimate instance a set of hardware interfaces.

Furthermore, ros_control offers different types of controllers (single joint, trajectory, differential drive, gripper controllers, etc in a separate package named "ros_controllers") that claim the resources located inside the robot abstraction and if two controllers require the same resource, the RobotHW class also manages the resource conflict. Each controller has two interfaces: on one side a JointCommandInterface that should match the corresponding hardware interface of the claimed resource and on the other side a regular ROS interface, which is actually a ROS topic that other nodes can use to publish control commands. These publishers can be third-party actors like the MoveIt! motion planning library, a terminal, an external navigation program, or just a regular node, so for example if a float number with a value of 1,5 is sent to the topic of a position_controller, it will try to control the associated resource, that is, to move a certain joint to a position of 1,5 radians.

On the other hand, there exists a controller manager that is in reality who is aware of the available hardware resources and assigns them to the controllers, it also enforces the resource conflict policy (first-come, first-served) and lists, (un-)loads, stops or switch controllers when it is told to do so. Finally, in ros_control there is a pair of additional interfaces: a joint limits interface to make sure that controllers will not damage the joints and a transmission interface to account for typical mechanical transmissions (simple reducer, four bar linkage, reducer) and the mapping of commands between the actuator and joint domains. Both can be easily included in the URDF of the robot as will be seen in the next chapter.

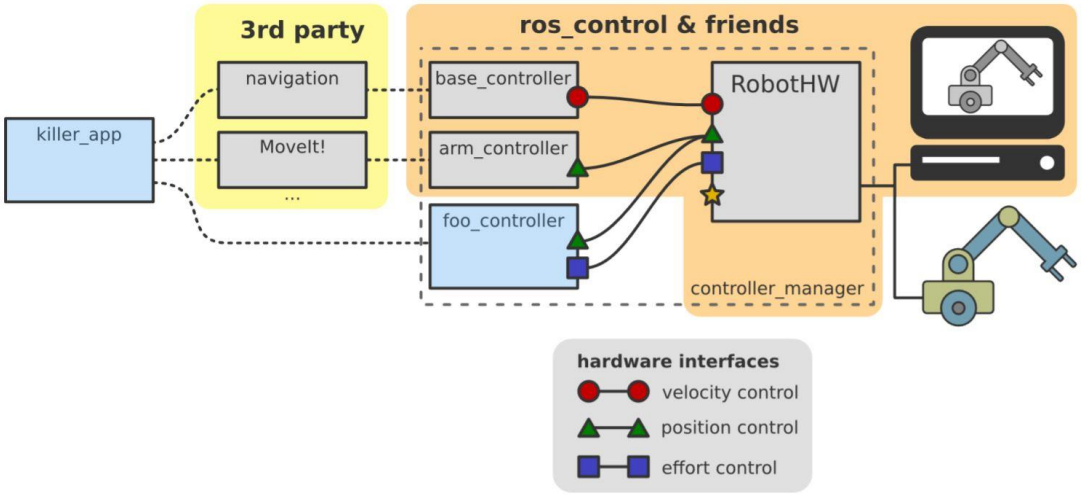


Fig. 5.4: ros_control overview scheme

5.2 Gazebo

Gazebo is a free 3D robotics simulator that allows to design virtual models of robots, simulate a wide range of sensors, objects and their interaction, and replicate realistic complex indoor/outdoor environments for rapid testing of algorithms [23]. The initial creators of Gazebo, Dr. Andrew Howard and his student Nate Koenig, started its development at the University of Southern California in 2002. Seven years later, John Hsu from Willow Garage achieved a promising approach towards ROS and since then it has become the first option of choice for ROS users that need a productive simulation tool. In 2012 the Open Source Robotics Foundation (OSRF) took the leadership of the project, and as a result now the integration between Gazebo and the ROS ecosystem is even more satisfactory than ever. In fact, OSRF utilized Gazebo in 2013 to set up the Virtual Robotics Challenge of DARPA (the Defense Advanced Research Projects Agency of the USA). In Fig 5.5 the GUI of the simulator is illustrated.

As it happens with ROS, Gazebo is backed up by a large active community (has also a dedicated Q&A website to share knowledge), which is a guaranty of great future support for a long period of time. Gazebo has also much more documentation available than other simulators, so these two are primarily the reasons why it was chosen along with ROS for this project. Perhaps V-REP could have been a good alternative because according to Nogueira [24] "…V-REP is a more intuitive and user-friendly simulator, and packs more features.", however "Gazebo is more integrated into the ROS framework and is an open-source solution which means it allows for complete control over the simulator." Multiple versions have been released, the current one is Gazebo 7.1, but the project was done with the 6.0. It is important to note that Gazebo runs best on Linux systems like Ubuntu; there is also a way to install it on Mac OS X, and for Windows the compilation does not work yet, but is in the roadmap of the eighth edition.

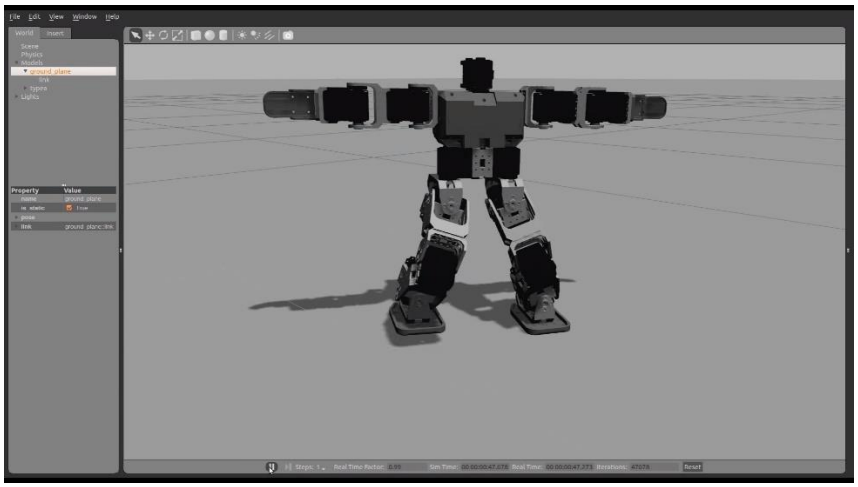


Fig. 5.5: Bioloid in the Gazebo simulator

5.2.1 Gazebo features

The latest versions of Gazebo present logically more advanced functionalities than the previous ones, but in broad terms these are:

- High performance dynamics: simulations can be tested with up to four different physics engines namely ODE, Bullet, Simbody and DART.
- Sophisticated 3D graphics: realistic rendering of shadows, textures, lighting of the items inside the scenario are provided by OGRE.
- Sensors and Noise: data of multiple sensors like contact, Kinect, laser range sensors and bi-/tri-dimensional cameras can be generated with noise if needed.
- Plugins: they expose a direct access to the API of Gazebo for users that wish to implement custom functionalities for environmental, sensor or robot control.
- Online Model database: contains simulated robots like the Pioneer 2 DX, Robonaut, Turtlebot among others and some ordinary objects for download.
- TCP/IP transport: simulations can run on remote servers, and interface to Gazebo through socket-based message passing using Google Protobufs.
- Command Line tools: make easier the observation, interaction and control of the simulation without making use of the GUI.

5.2.2 Gazebo components

Whenever a simulation is running, a group of elements involved in the execution process can be clearly identified:

- Gazebo server: (gzserver) it is the core of the application in charge of parsing a world description file to simulate it using the physics and sensor engine. It does not output any graphics though.
- Graphical client: (gzclient) makes a connection with an active gzserver for a visual representation of the parsed world. The user can alter with this GUI the simulation (play, pause, resume, restart, insert/delete models, etc)
- Plugins: as already mentioned, these are a fragments of code that provide a simple and optimal interface to Gazebo, which loads most of them upon start, but they can also be inserted in world /model files.
- World files: they have a .world extension and contain all the elements (robots, lights, sensors, and fixed objects) that gzserver should interpret for being simulated. The file's structure is written in SDF (Simulation Description Format).
- Model files: describe exclusively one single model (a robot or a regular object) also in SDF syntax. A world file may include several model files to simplify its length, so in this way models can be used again by other world files.
- Environment variables: are used by Gazebo establish connections between the client and server and to locate files: models, the simulator master URI, model database URI, plugins and other resources like world and media files.

5.3 MATLAB

Matlab (abbreviation of MATrix LABoratory) is a commercial mathematics software developed by the company Mathworks [25] that provides an integrated development environment (IDE, shown in Fig 5.6) with its own programming language of the same name (also simply known as “m language”). It is available for the Windows, Linux and Mac OS X platforms.

Some of its outstanding capabilities are the manipulation of matrices, data plots and graphs, countless amount of functions for solving many types of equations, implementation of algorithms, creation of custom graphical interfaces, etc. It’s definitely a very popular software suite within academic and research centers as well as for industrial applications. The version employed for this project is the 2016a.

5.3.1 Simulink

Simulink [26], which runs above Matlab even though is marketed as an individual product, is a graphical programming environment to model, simulate and analyze dynamics systems with a certain abstraction degree of the actual physical phenomena, for example in the fields of electronics digital signal processing (DSP), control engineering, and of course robotics. Therefore, it is a programming tool of higher-level than the interpreted language of Matlab. Controllers can be implemented in Simulink by linking different diagram blocks categorized into libraries. Another of its significant characteristics for this project is the possibility to include also blocks with regular Matlab code inside the models designed, which are saved with a .mdl extension.

5.3.2 Robotics System Toolbox

Mathworks offers a set of complementary add-on products to extend the default capabilities of Matlab and Simulink, called toolboxes. For this project is extremely relevant the Robotics System Toolbox [27], introduced since the 2015a version, that acts basically as a native interface between Matlab/Simulink and ROS (in the end this means Gazebo as well) bringing together the best of both worlds: the computing power of Matlab (control of motors, computer vision algorithms, state machine diagrams…) and the extraordinary features and modularity of the ROS middleware.

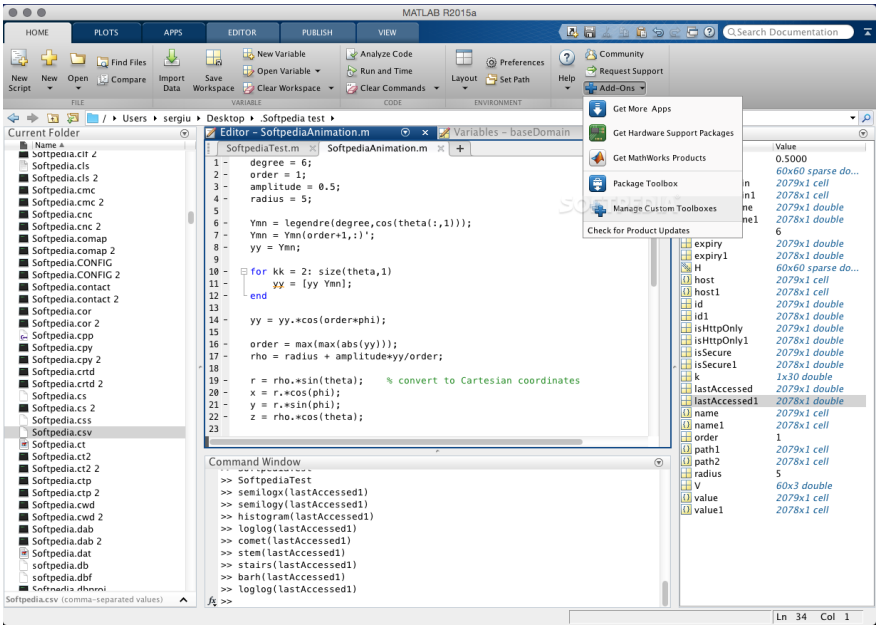


Fig. 5.6: MATLAB integrated development environment

5.4 Blender

Blender is an open-source creation suite for design and modeling of objects in 2D and 3D space aimed at artists, media and graphic design professionals [28]. Moreover, it is a conventional tool within the ROS community for editing the overall appearance properties of the robot and world meshes (size, color, textures, setting of the mesh origin). The Blender GUI appears in Fig. 5.7. It was first launched as a finished product in 1994 by the enterprise "Not a Number" (NaN) with the idea of being a free-license program from the very beginning. Nowadays the Blender Foundation of the Netherlands continues to improve it in each iteration release. For this project the current version 2.77a was utilized. The main functionalities of Blender that deserve to be mentioned are: images representation and post-processing, a completely integrated design suite, a wide range of essential tools for the creation of 3D content that includes sculpting, UV mapping, texturing, characters animation, particles simulation, scripting, rendering, composition, post-production and creation of videogames. Blender relies on OpenGL for the rendering of graphics; is available for the three operating system architectures and can be readily customized with python scripts to broaden its use cases. Other bonus points of this CAD program is the amount of tutorials and manuals found online as well as the helping attitude of the questions forum.

Fig. 5.7: Blender graphical interface

5.5 Meshlab

MeshLab is an open source application that allows the processing and editing of unstructured 3D triangular meshes., and comes with a set of tools for filtering, cleaning, healing, inspecting, rendering and converting this kind of meshes. Meshlab is heavily based on the VCG library developed at the Visual Computing Lab of an Institute of the National Research Council of Italy. The MeshLab project began in late 2005 as a part of the FGT course of the Computer Science department of University of Pisa. In recent years FGT students have worked to implement more and more features [29].

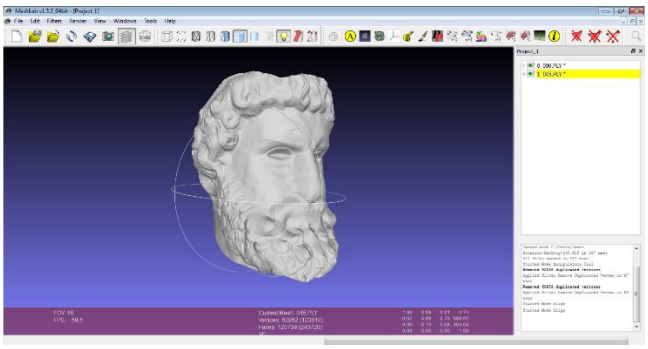


Fig. 5.8: MeshLab graphical interface

Chapter 6

Project elaboration

From now on, the stages that were followed to reach the ultimate objective of this project, which is the implementation of a development environment for the Raider mini-humanoid robot, will be explained, starting first with a sketch of the component-based ensemble and then delving into the peculiarities of each of the elements involved.

6.1 Development environment overview

The development environment has been created according to the guidelines of the component-based modeling approach mentioned in Section 4.2. There are clearly three independent of each other and distinct modules in the ensemble, as depicted in Fig. 6.1: On one side Gazebo will be used to simulate the mini-humanoid robot itself and the robot scenario; on the other side, a controller will be designed with MATLAB/Simulink to command the robot and in between the ROS middleware will act as a bridge that connects them to make possible the interaction controller-simulator.

Note that the three modules are loosely coupled, so each of them can be individually replaced, modified or deleted if necessary without affecting the others. This intrinsic flexibility of a component-based model is exactly one of the objectives laid out in the second chapter. Alright, but…how do Gazebo and MATLAB/Simulink interface in turn with ROS? Short answer: they become nodes inside a ROS network and as every node else, they can publish or subscribe to topics for data exchange. This will be anyway discussed more in depth in the next pages.

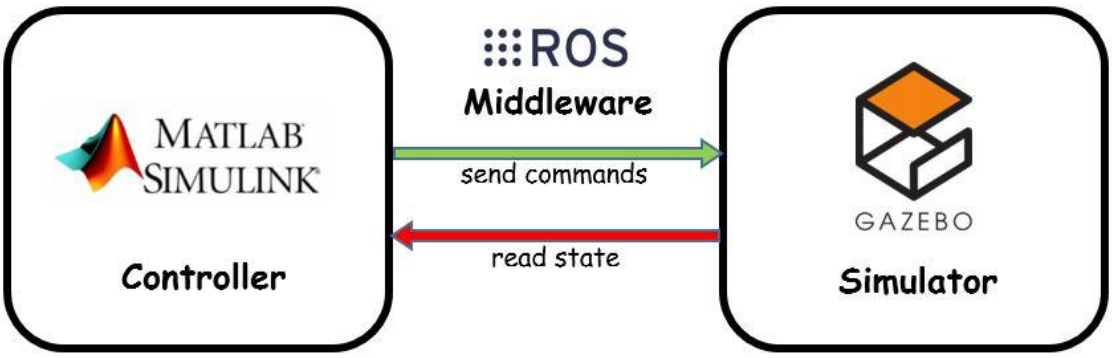


Fig. 6.1: Development environment overview scheme

6.2 Simulation of the mini-humanoid robot

At this point, before going down to the business, it is worth mentioning that initially the simulation of the robot was not an objective. Patricia Muñoz, a master student with whom there has been a narrow collaboration in the last months, was doing her master thesis about simulating all aspects of the real Raider (appereance, physical properties, and sensors) and all scenarios of the CEABOT competition, at the same time as this project was being developed. Since she shared the same thesis director, it was agreed that she would have to provide the author of this document with a preliminary version of the simulated robot suitable to be controlled. The robot looked visually fine, however, when trying to command the robot, it was completely unstable due to an inaccurate setting of its dynamic properties. Specifically, the center of mass of each link was not located within the piece volume, but rather far away, probably because the meshes were exported wrongly from the CAD application with its corresponding origins not centered in the (0, 0, 0) coordinates and also its inertia tensor values were of orders of magnitude too big for such a small robot.

These problems sadly could only be solved by building the virtual robot model in URDF again from scratch and it was then in that moment when the author of this project assumed on his own initiative the task of simulating Raider, taking as a basis the previous work done partially well by Patricia.

6.2.1 Gazebo-ROS integration

After installing Ubuntu 14.04 on the computer, the first step was reading the documentation about ROS and Gazebo to become familiar with them, read the tutorials ([30], [31]) and know if there are any compatibility issues. In [32] it is indeed clarified which combination of ROS/Gazebo versions to use, so it was decided to go ahead with ROS Indigo and Gazebo 6.0 in tandem. It is important to bear in mind that they should be installed in that particular order: first ROS, then Gazebo, because if one wishes to use both, the installation procedure of Gazebo is not the same as installing its individual standalone version. Instead a metapackage (a set of packages) named “gazebo_ros_pkgs”, which contains the core of the simulator plus wrappers with the required ROS interfaces to simulate a robot in Gazebo; needs to be installed. In this way, just by installing “gazebo_ros_pkgs”, the integration of Gazebo within the ROS ecosystem is achieved seamlessly and the user is not forced to take any more configurations steps. The metapackage “gazebo_ros_pkgs” includes three packages:

- gazebo_ros: wraps gzserver and gzclient by using two Gazebo plugins that provide interfaces for messages, services and dynamic reconfigure (this term refers to the modification of the world physics when the simulation is running).
- gazebo_msg: messages and service data structures to interact with Gazebo transmitting information from ROS.

- gazebo_plugins: robot independent Gazebo plugins (*e.g.* for simulation of sensors).

As already outlined, two plugins are located in the gazebo_ros package:

- gazebo_ros_api_plugin: initializes a ROS node called "gazebo". It integrates the ROS callback scheduler (message passing) with Gazebo's internal scheduler to provide the ROS interfaces. This ROS API enables a user to manipulate the properties of the simulation environment over ROS, as well as spawn and introspect on the state of models in the environment.
- gazebo_ros_paths_plugin: simply allows Gazebo to find ROS resources, that is, resolving ROS package path names.

The node "gazebo" subscribes to a couple of topics where data arrives to change the state (pose/twist) of models/links within the simulated environment. This node also publishes the current state of model/links to topics other nodes can listen to, and exposes some useful services to interact with the simulation such as spawn/delete models, setters/getters of the objects state and properties, force control of links/joints and simulation control (pause/unpause pyhisics updates, reset simulation and reset world). Especially interesting is the published parameter "/use_sim_time", which is set true automatically when gazebo_ros is started to notify ROS that simulation time is being published to the topic "/clock", so that both Gazebo and ROS can operate in a synchronized manner [33]. The mentioned packages are listed in Fig 6.2.

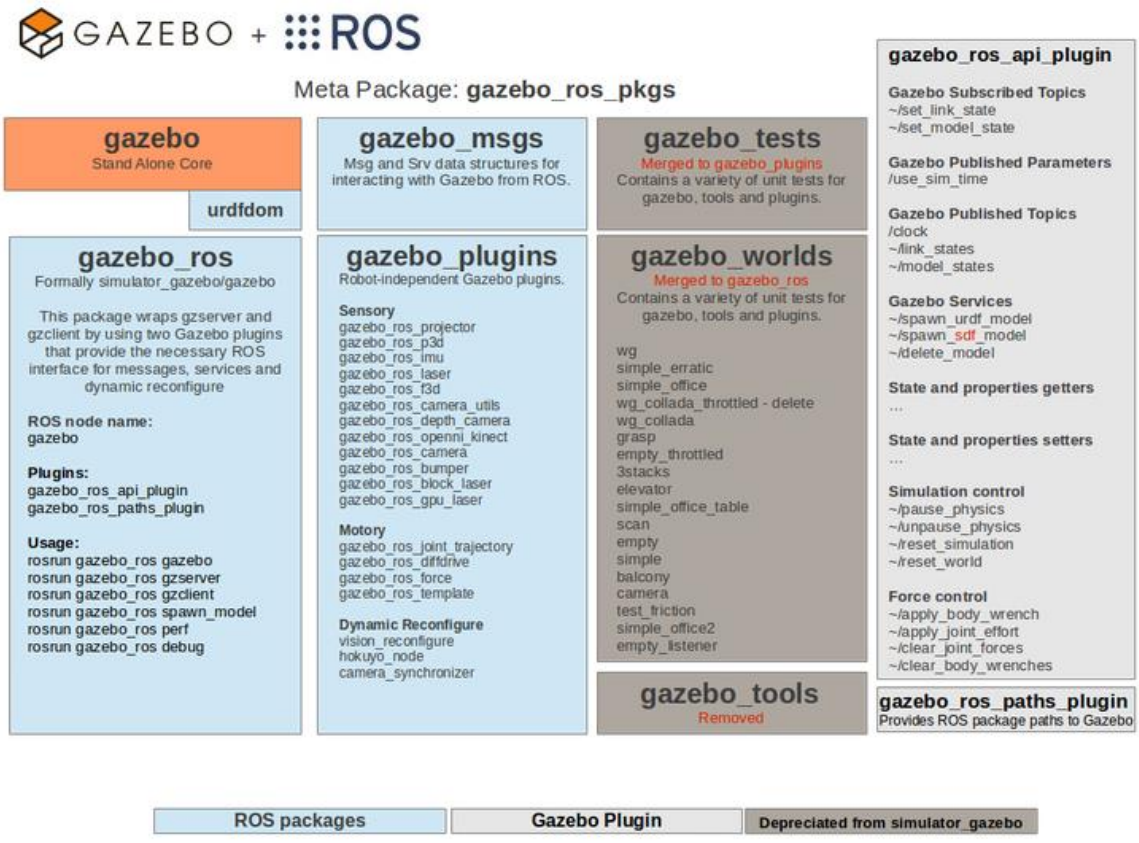


Fig. 6.2: Packages required for the integration of Gazebo within ROS

6.2.2 Files hierarchy

Once ROS is installed, one should create in any directory, for instance in "/home/user/" a workspace for the ROS compilation tool "catkin". In this project, the name of this workspace is "catkin_ws" and in there (actually in the "/src" subfolder) must reside all the files produced by the user himself or those packages downloaded that need to be compiled from source. The recommend file hierarchy, shown partially in the next figure between "*", has been adopted, although it is not strictly mandatory. Instead of doing it by hand, it was much better to download and compile with the instruction "catkin_make" the example code of the repository [34] that stores "rrbot", a simple test robot composed of three bars; copy the folders structure and then change the names of the packages and files as well as its content accordingly to customize them to be used for the mini-humanoid Raider.

```
../catkin_ws/src
    /raider_description  *package for robot description files*
        package.xml      *defines package properties (e.g. author)*
        CMakeLists.txt   *required for compilation purposes*
        /launch          *contains roslaunch files*
            raider_rviz.config   *configurates default view in Rviz*
            raider_rviz.launch   *launches Rviz to display the robot*
        /urdf            *contains the urdf file*
            Raider.urdf          *model description of the robot*
        /meshes          *contains the different pieces of the robot*
            Head.dae             *cad file with .dae extension*
            Chest.stl            *cad file with .stl extension*
            ...
            ...
    /raider_gazebo       *package for robot and world simulation in Gazebo*
        package.xml      *defines package properties (e.g. version)*
        CMakeLists.txt   *required for compilation purposes*
        /launch          *contains roslaunch files*
            raider_world.launch  *launches the world to be simulated*
        /worlds          *contains world files*
            obstacle_race.world  *file with .world extension*
        /models          *contains objects to be simulated*
            /floor               *folder for a particular model*
                Floor.dae                *cad file with .dae extension*
                floor_texture.jpg        *texture as .jpg image*
            ...
            ...
```

Fig. 6.3: Fragment of the files hierarchy

From the previous hierarchy, most of the files are self-explanatory, however launch files deserve a remark. They are very common in ROS and represent a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters. The command "roslaunch" is used in conjunction with these launch files. This can be done by either specifying the package that contains the launch

files followed by the name of the launch file, or by specifying the file path to the launch file. For example, during the intermediate stages of the construction of the Raider model in URDF, the “raider_rviz.launch” file that calls Rviz to display the robot was executed once in a while to check that there were no misalignments.

6.2.3 Preparation of the meshes

As explained at the beginning of this section, a preliminary version of the simulated robot was given to the author of this project by a female master student, but there was a problem with the meshes as they were not exported correctly from the origin of the CAD editor employed by her (Freecad) until that moment and they also had no colors associated, so these pieces needed to be painted and its origin had to be fixed again. In order to accomplish this task, Blender came in handy as it lets apply colors as textures to the collada files (with extension. dae), which are the cad files preferred to reproduce the external appearance of objects in ROS/Gazebo.

Thus, all the pieces of Raider were imported to the scene of Blender. They were applied colors to make them look to some extent like the pieces of the real robot at first glance, although not exactly equal. Also, because the robot has too many pieces, the ones that are in contact, but don’t move relative to each other, were merged into a new single (and bigger) piece. For example, the chest chassis contains a compass, a couple of servos, the controller board on its back, etc., but all of them are firmly attached, so they were aggregated in a single piece named simply “Chest”. The origin of each new piece was then set to the spot where it is joined to the next mobile part to make shorter the subsequent definition of the URDF as will be discussed in the next page. Finally, the new compact pieces were placed at the origin of the scene and then exported one by one again as .dae files. Fig 6.4 illustrates the full Raider in the Blender environment.

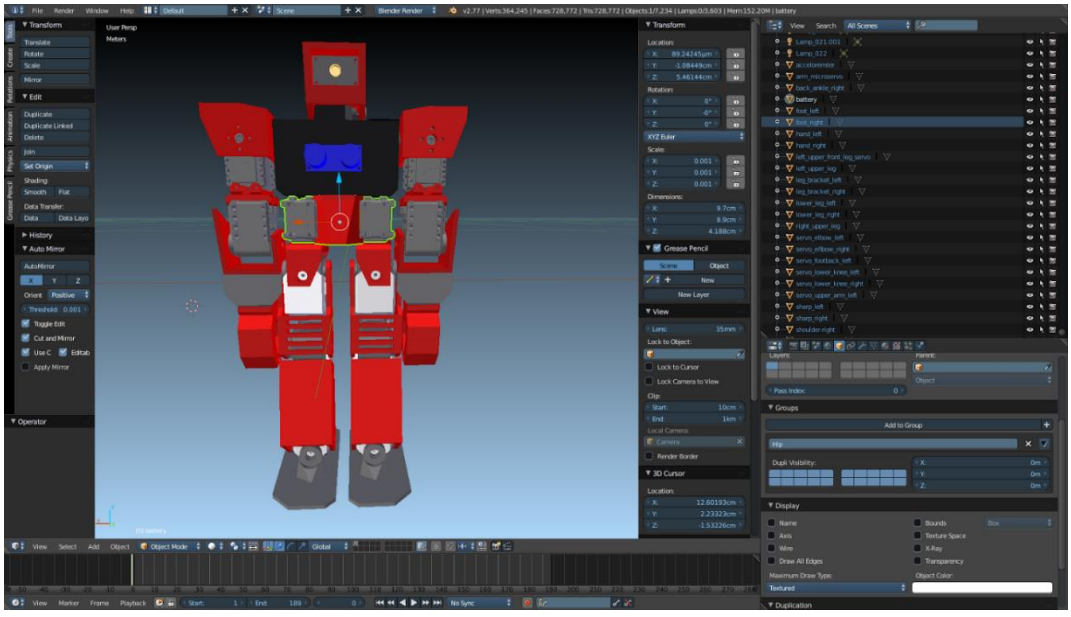


Fig. 6.4: Pieces of Raider in Blender at an intermediate stage of the coloring process

6.2.4 Definition of the URDF-described Raider

In order to take advantage of the multiple useful features provided by ROS, it is necessary to build first a virtual model representation of the robot specified in URDF (Uniform Robot Description Format), which is based on the XML language [35]. A robot is treated in URDF basically as a collection of links connected through joints, as it is shown in Fig. 6.5, where there can only be one root link. A vague description of the important characteristics of the URDF link and joint components is presented next.

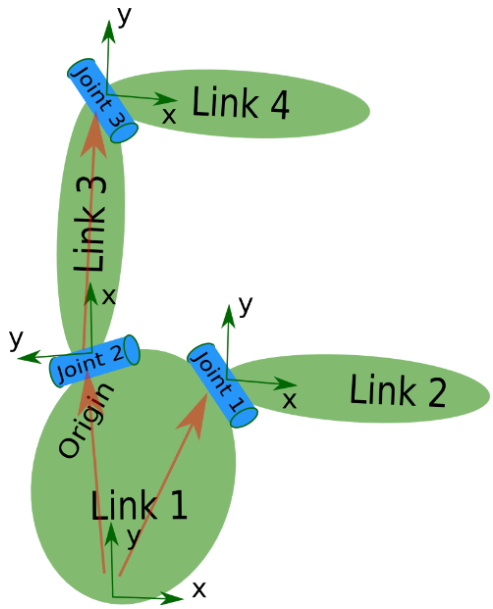


Fig. 6.5: Sample robot structure

6.2.4.1 Link

- Attributes: only a name is required
- Elements:
  - Inertial: contains information about how heavy the link is (the mass), the location of the center of mass with respect to the link reference frame, and the 3x3 rotational inertia matrix represented in this inertial frame.
  - Visual: indicates the origin of the shape that represents the external appearance of the link with respect to link reference frame. This shape can be identified as a simple box, sphere or cylinder with a rgb material color or it can also be complex mesh.
  - Collision indicates the origin of the shape for collision detection with respect to the link reference frame. This shape can be identified as a simple box, sphere or cylinder, or a complex mesh.

These elements are displayed in Fig 6.6. So, as we see three origins inside a link have to be specified for each element, and they do not necessarily need to coincide. Also note that the collision properties can be different from the visual properties of a link, for example, simpler collision geometries are often used to reduce computation time.

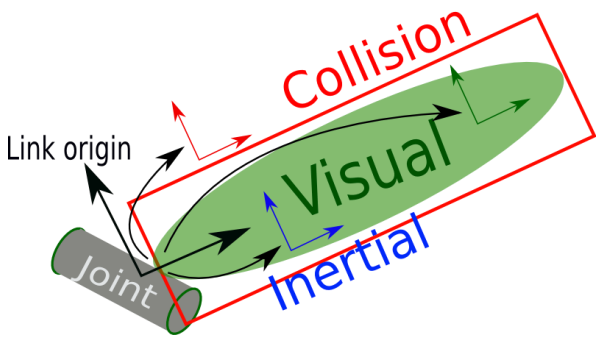


Fig. 6.6: URDF link elements

6.2.4.2 Joint

- Attributes: the name of the joint and its type (revolute, fixed, prismatic, etc.)
- Elements:
  - Origin: this is the transform from the parent link to the child link. (x,y,z translation coordinates and roll, pitch, yaw rotation angles). The joint is located with respect to parent link and shares the origin of the child link.
  - Parent: specifies the parent link name
  - Child: specifies the parent link name
  - Axis: This is the axis of rotation, translation, or surface normal for revolute, prismatic or planar joints respectively.
  - Dynamics: details physical properties of joint such as static friction and damping, particularly useful for simulation.
  - Limits: only applicable to revolute and prismatic joint. Indicates the lower and upper and joint bounds as well as the maximum effort that can be exerted on the joint and the maximum joint velocity.

The parent and child link of a joint and its reference frames are shown in Fig. 6.7. Note that since the child link origin is equivalent to the joint origin, a certain joint is actually always located with respect to the previous joint.

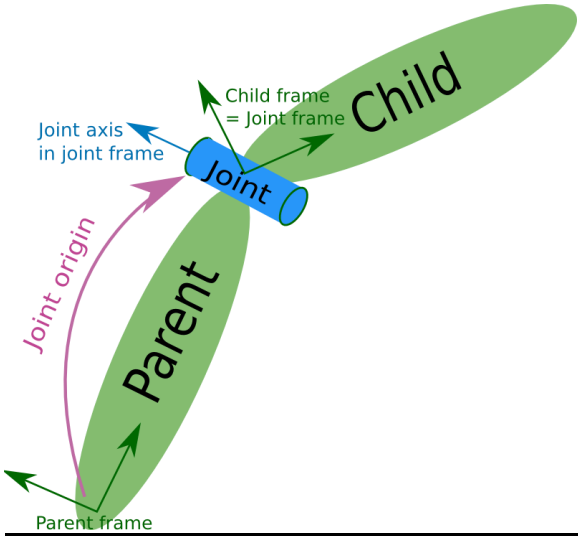


Fig. 6.7: URDF joint elements

6.2.4.3 Raider URDF highlights

The file Raider.urdf defining the robot has a quite long extension, therefore only little fragments of it that are of special emphasis will be commented. A complete diagram of all the parent-child relations can be checked in the Appendix. It is important to say that the mass values of this URDF have been taken from the data collected by Patricia Muñoz, who obtained them from an estimation done by a 3D printer program based on the mesh volumes and the density of a plastic material. Also, remember that all mesh files were exported with its origin placed at the spot where they should be joined, which allowed to put always inside every <visual> and <collision> elements of the links the tag <origin rpy = “0 0 0” x y z =”0 0 0”/> because as can be seen in Fig. 6.7 the joint frame equals the link frame, so in this case there is no translation nor rotation of the meshes relative to that point.

In Fig. 6.8 it is observed that two links named “hip” and “chest” are specified, and these are connected by a joint named “middle_hip_twist”. The parent link “hip” has a mass of 0.354 kg and is represented visually by a Hip.dae file and for collision detection by a Hip.stl. Something similar happens with the child link “chest”, it weighs 0.4759 kg and its geometries are Chest.dae for visualization and Chest.stl for collision

```
12  <link name="hip">
13    <collision>
14      <origin rpy="0 0 0" xyz="0 0 0"/>
15      <geometry>
16        <mesh filename="package://raider_description/meshes/Hip.stl" scale="1 1 1"/>
17      </geometry>
18    </collision>
19    <visual>
20      <origin rpy="0 0 0" xyz="0 0 0"/>
21      <geometry>
22        <mesh filename="package://raider_description/meshes/Hip.dae" scale="1 1 1"/>
23      </geometry>
24    </visual>
25    <inertial>
26        <origin xyz="0.0000122 -0.0121975 0.0013754" rpy="0 0 0"/>
27      <mass value="0.354"/>
28        <inertia ixx="2.399113558e-1" ixy="7.449718907e-4" ixz="3.800877e-4" iyy="2.74636168e-1" iyz="4.409017312e-4" i
29    </inertial>
30  </link>
31
32  <joint name="middle_hip_twist" type="revolute">
33   <dynamics damping="0.2" friction="1"/>
34   <limit upper="2.618" lower="-2.618" velocity="5.6" effort="3"/>
35    <parent link="hip"/>
36    <child link="chest"/>
37    <origin xyz="0 0.01068 0.0558014"/>
38    <axis xyz="0 0 1"/>
39  </joint>
40
41  <link name="chest">
42    <collision>
43      <origin rpy="0 0 0" xyz="0 0 0"/>
44      <geometry>
45        <mesh filename="package://raider_description/meshes/Chest.stl" scale="1 1 1"/>
46      </geometry>
47    </collision>
48    <visual>
49      <origin rpy="0 0 0" xyz="0 0 0"/>
50      <geometry>
51        <mesh filename="package://raider_description/meshes/Chest.dae" scale="1 1 1"/>
52      </geometry>
53    </visual>
54    <inertial>
55        <origin xyz="-0.00022088 -0.01276272 -0.00348415" rpy="0 0 0"/>
56      <mass value="0.4759"/>
57        <inertia ixx="4.223702652e-1" ixy="-3.301078629e-3" ixz="2.090821448e-3" iyy="6.174246752e-1" iyz="-1.577048904
58    </inertial>
59  </link>
```

Fig. 6.8: Fragment of Raider.urdf

Actually all joints of the URDF, have in common all properties, except of course for the name, origin and axis of rotation. In the preceding fragment the joint "middle_hip_twist" is of type revolute with dynamics properties of 0.2 N·m friction and 1 N·m·s/rad damping. These values were chosen arbitrarily, but within a margin of realism to provide the joints with some movement opposition and for stability purposes. The <origin x y z ="0.0168 0 0.055804"/>, found approximately with the cursor in Blender, determines its location with respect to the parent link. Its axis of rotation is around the z axis; it has upper and lower limits of 2.618 and -2.618 radians (so in total they can rotate 5,236 rad (almost 300 degrees), a limit effort of 3 N·m and it can reach a maximum angular velocity of 5.6 rad/s. Each numerical value must be introduced in the URDF in standard units of the International System (kg, m, N, rad, s, etc.…).

These last joint properties try to emulate the real operation of the Dynamixel AX-12A servos the real Raider counts with. According to their datasheet, these servos can make a turn of up to 300°, provide a stall torque of 1.5 N·m and can reach a maximum speed of 114 rpm (approx. 11,94 rad/s), yet it was decided to copy the effort and velocity limit values from other URDF examples of mini-humanoid robots ([36], [37]) that perform reasonably good in simulation and work also with the Dynamixel AX-12A servos, such as the Bioloid and the Darwin-OP robots, both produced by Robotis.

The trickiest part of the modeling process was for sure how to obtain kind of suitable inertial parameters. This was accomplished in the following manner: each mesh was imported to MeshLab to compute the geometric measures of the piece as recommended in a tutorial at the Gazebo website [38], but as the pieces were of small size, the values obtained for the inertia matrix were also so small that MeshLab only could show an output of zeros with its maximum precision of 6 digits. To overcome this situation, the tutorial suggested applying a scale transformation to the pieces to get significant and bigger inertia values that should be however scaled back before putting them into the URDF inertial tags. Thus, the meshes were magnified by a scale of 100 in MeshLab (only for computation purposes, they were not saved like that). This program assumes a unitary density for the computation of the inertia, thus since the dependency of volume with the scale is to the cube and the inertia varies with the scale to the square, the inertia varies in total to the fifth. Hence, to scale back the large inertias obtained, they were divided by 100 to the fifth and divided by the real volume and multiplied by the mass of the object to undo the unitary density assumption. Nevertheless, the values obtained by this method were surprisingly too much small in comparison to the found ones in the Bioloid and Darwin URDF files, making the robot shake when tried to be controlled, so they were multiplied once more by 1000 so that they were of an order of magnitude comparable with the inertias present in the URDF files of the above mentioned mini-humanoid robot examples. Conversely, the location of the center of mass was not a big deal, it sufficed copying and pasting the value calculated by MeshLab before applying the scale. An example of the output window after the computation of these parameters in MeshLab is displayed in Fig. 6.9.

```
Mesh Bounding Box Diag 15.222200
Mesh Volume is 197.247818
Mesh Surface is 1070.988892
Thin shell barycenter -0.037237 -1.940958 -0.291330
Center of Mass is -0.023258 -1.284886 -0.371095
Inertia Tensor is :
| 1755.578613 -19.991816 9.871661 |
| -19.991816 2558.695557 -63.275898 |
| 9.871661 -63.275898 2950.035889 |
Principal axes are :
| 0.999680 -0.022930 0.010687 |
| 0.024319 0.987425 -0.156205 |
| -0.006970 0.156415 0.987667 |
axis momenta are :
| 1755.023438 2549.136475 2960.150146 |
```

Fig. 6.9: Geometrical properties computed by MeshLab of the Chest.stl file, after being scaled by 100

Moreover, MeshLab was helpful for another procedure that was the simplification of the collision meshes. The visual appearance of the links can be represented by a mesh with a lots of details, since its rendering does not require an excessive graphics power, but for collisions detection simpler geometries are much better to reduce the stress on the computer processor, computational cost and time. Consequently, all the pieces were exported from Blender also as .stl files (which are less detailed than .dae files) and then imported into MeshLab to be applied a filter (the so called Quality Edge Colapse Decimation) for reducing its number of faces and vertices to about 0.3 percent.

Now, a very relevant aspect in the simulation of mini-humanoid robots like Raider is a stable contact of the feet with the ground, so that it does not slip and fall frequently. In Fig. 6.10 the parameters associated with each foot are shown.

```
817
818 <gazebo reference = "left_foot">
819     <mu1>9000</mu1>
820     <mu2>9000</mu2>
821     <kp>1000000.0</kp>
822     <kd>10.0</kd>
823     <minDepth>0.001</minDepth>
824     <maxContacts>1</maxContacts>
825 </gazebo>
826
827
828 <gazebo reference = "right_foot">
829     <mu1>9000</mu1>
830     <mu2>9000</mu2>
831     <kp>1000000.0</kp>
832     <kd>10.0</kd>
833     <minDepth>0.001</minDepth>
834     <maxContacts>1</maxContacts>
835 </gazebo>
836
```

Fig. 6.10: Feet contact parameters in Raider.urdf

It can be observed that those parameters are enclosed within a <gazebo> tag and the reason behind this fact is that they can only be interpreted by Gazebo. The gazebo tags extend the description possibilities of the URDF for outside-ROS simulation of robots. It is enough to specify in the reference field the link that should be characterized by those parameters. In this case, the feet contact parameters have the following meaning:

- mu1 and mu2: Friction coefficients μ for the principal contact directions along the contact surface as defined by the Open Dynamics Engine (ODE).
- kp and kd: Contact stiffness k_p and damping k_d for rigid body contacts as defined by ODE (ODE uses erp and cfm but there is a mapping between erp/cfm and stiffness/damping
- minDepth: minimum allowable depth until contact correction impulse is applied.
- maxContacts: Maximum number of contacts allowed between two entities. This value overrides the max_contacts element defined in physics.

These values were taken again from the Bioloid and Darwin examples as they seemed to deliver good results for the stabilization of the robot. No wonder, since Raider is almost the same size and weight as the other mini-humanoids. An interesting thing about the gazebo tags meant to be used for links is the <selfCollide> property, which enables collision detection between two non-consecutive links of the robot. By default, Gazebo only detects collision between the robot and an external object, not between the own links of the robot itself. This feature was tested, but it was discarded because it lowered the real time factor of the simulation too much due to the huge computational cost, it made the robot sometimes wobble, and after all, with the design of a good controller, self-collision detection was not much of a help. The gazebo tags can also refer to joints and to the whole robot if no reference is specified; this is especially useful for the inclusion of plugins and sensors.

So, the description of the robot has come almost to the end. The explanation of the transmission tags and sensor plugins is still left, but let's focus for a moment on the launch file to execute the Rviz visualization tool in order to be able to inspect the appearance of the robot; the content of such file is indicated below in Fig. 6.11.

*raider_rviz.launch ×

```
1 <?xml version="1.0"?>
2
3 <launch>
4   <param name="robot_description" textfile="$(find raider_description)/urdf/Raider.urdf" />
5
6   <!-- send fake joint values -->
7   <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
8     <param name="use_gui" value="TRUE"/>
9   </node>
10
11  <!-- Combine joint values -->
12  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher"/>
13
14  <!-- Show in Rviz   -->
15  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find raider_description)/launch/raider.rviz"/>
16
17 </launch>
```

Fig. 6.11: Content of the raider_rviz.launch file

What happens when this file is launched is the following: if no ROS master has been created, it starts a ROS network with a master; then in the parameter server, the path to Raider.urdf file is stored under the parameter “robot_description”, so that it can be parsed by other components of the middleware, Finally, three nodes are generated:

- “robot_state_publisher”: precisely reads the URDF file specified in the parameter “robot_description” and the joint positions arriving at the topic “/raider/joint_states” to calculate the forward kinematics of the robot and publish the results via the library for transformation of coordinates between frames (TF).
- “joint_state_publisher”: is a modest GUI with sliders for sending fake joint values to the topic “/raider/joint_states”. In other words, it serves to change the pose of the robot in Rviz (without any physics involved).
- “rviz”: this node opens the Rviz tool and receives the data of transformations between the different robot frames for a complete visualization of the robot components (links, joints) in Rviz.

In Fig. 6.12 it is displayed the terminal output after having entered in the command line “roslaunch raider_description raider_rviz.launch” and in Fig. 6.13 appears the final version of Raider fully assembled and visualized in Rviz.

```
xavidz@xavidz:~$ roslaunch raider_description raider_rviz.launch
... logging to /home/xavidz/.ros/log/7239e37e-839f-11e6-8303-185e0fbd4a2f/roslaunch-xavidz-6024.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://xavidz:38891/

SUMMARY
========

PARAMETERS
 * /joint_state_publisher/use_gui: True
 * /robot_description: <?xml version="1....
 * /rosdistro: indigo
 * /rosversion: 1.11.20

NODES
  /
    joint_state_publisher (joint_state_publisher/joint_state_publisher)
    robot_state_publisher (robot_state_publisher/state_publisher)
    rviz (rviz/rviz)

auto-starting new master
process[master]: started with pid [6036]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 7239e37e-839f-11e6-8303-185e0fbd4a2f
process[rosout-1]: started with pid [6049]
started core service [/rosout]
process[joint_state_publisher-2]: started with pid [6060]
process[robot_state_publisher-3]: started with pid [6067]
process[rviz-4]: started with pid [6068]
```

Fig. 6.12: Terminal output of "roslaunch raider_description raider_rviz.launch"

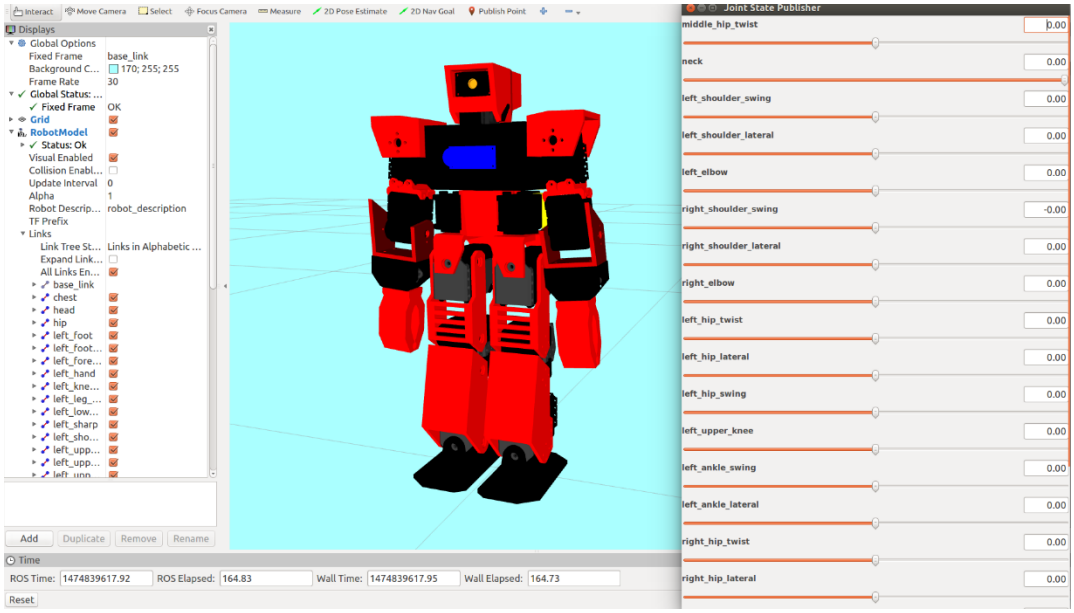


Fig. 6.13: Final appearance of Raider visualized in Rviz

In order to manipulate a simulated Raider, the packages "ros_control", "ros_controllers" and "gazebo_ros_control" were installed (the last one downloaded to the catkin_ws from the github repository of the gazebo_ros_control project and compiled manually from source with the catkin tool because there is bug in the package from the Ubuntu repository that makes Gazebo crash at start). The first two packages were discussed in Section 5.1.3. The package "gazebo_ros_control" basically consists of a plugin with the same name to be inserted inside the URDF for parsing the information contained in the <transmission> tags, it also loads the appropriate hardware interfaces and the controller manager. Additionaly, the package contains in broad terms an implementation of the "RobotHW" class from "ros_control" named "RobotHWSim" that provides API-level access to read and command joint properties in the Gazebo simulator. However, the "RobotHWSim" is only a virtual class, so by itself it is not enough to drive a simulated robot, that's why the plugin provides also a "DefaultRobotHWSim" derived class that provides out-of-the-box control capabilities. The fragment of the URDF regarding this part can be seen in Fig.6.14.

Currently only simple transmissions are supported for simulation in Gazebo. The <joint> tag inside the transmission specifies the joint to be controlled and the <hardwareInterface> tag determines the control method (position, velocity or effort): The gazebo_ros_control plugin will attempt to get all of the information it needs to interface with a ros_control-based controller out of the URDF. This is sufficient for most cases, and good for at least getting started. The default behavior provides the following ros_control interfaces:

- hardware_interface::JointStateInterface
- hardware_interface::EffortJointInterface
- hardware_interface::VelocityJointInterface - *not fully implemented*

```
838  <!-- Import gazebo_ros_control plugin -->
839  <gazebo>
840    <plugin filename="libgazebo_ros_control.so" name="gazebo_ros_control_raider">
841      <robotNamespace>/raider</robotNamespace>
842      <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
843    </plugin>
844  </gazebo>
845
846  <transmission name="middle_hip_twist_tran">
847    <type>transmission_interface/SimpleTransmission</type>
848    <joint name="middle_hip_twist">
849      <hardwareInterface>PosVelJointInterface</hardwareInterface>
850    </joint>
851    <actuator name="middle_hip_twist_motor">
852      <hardwareInterface>PosVelJointInterface</hardwareInterface>
853      <mechanicalReduction>1</mechanicalReduction>
854    </actuator>
855  </transmission>
```

Fig. 6.14: gazebo_ros_control plugin and <transmission> tag in Raider.urdf

So, unluckily, the current default functionality of "DefaultRobotHWSim" only enables the control of joints by applying a torque to them (effort), which in the end implies a cumbersome process of tuning of PID controllers. It is impossible as well to run two controllers that have access to the same joint simultaneously because a resource conflict policy prohibits it. Thus, as the ultimate goal was finding an easy way to control Raider by simply specifying through controller topics target positions for all joints with a desired velocity, the files that contained the "DefaultRobotHWSim" definition were studied carefully to see if there was a chance for customization. After a deep analysis of the "default_robot_hw_sim.cpp" file located in "gazebo_ros_control/src" and reading of the Gazebo API functions for joint velocity control, the conclusion reached was that a working solution would be implementing a custom "PosVelJointInterface" inside the "DefaultRobotHWSim". that combines the position and velocity controllers in a single interface, but first the only-one-controller-only-one-joint policy needed to suppressed.

To do that, the package "hardware_interface" was downloaded to the catkin_ws to be modified and override the package installed from the Ubuntu repository. Inside the package "hardware_interface" there is a library called "robot_hw.h" that has a function called "checkForConflict", which returns true when two or more controllers compete for the same resource. The body of this function was changed to return always false. There should be probably a more elegant way of achieving the same, so that the whole process does not sound too much tedious at first sight. After that, in the "default_robot_hw_sim.cpp" file the piece of code of Fig. 6.15 was added (plus a few other lines to make it work, but that is the key portion and the file "default_robot_hw_sim.h" suffered slightly changes as well). Whoever that wishes to continue with this project has to compile from source with catkin these two customized packages ("hardware_interface" and "gazebo_ros_control") that will be available for download in the github repository that hosts all the files of this project [39]., otherwise the control simulation in Gazebo will not work at all.

*default_robot_hw_sim.cpp ×

```
370
371       // **PosVelJointInterface control algorithm**
372
373       case POSVEL:
374         {
375
376         double pos_delta_rad = (joint_position_command_[j] - joint_position_[j]);
377         bool goal_reached =  std::abs(pos_delta_rad) < allowed_error;
378         if(!goal_reached) {
379          double target_velocity = 0;
380          if(joint_velocity_command_[j] == 0)
381                 target_velocity =  sgn(pos_delta_rad) * 1;
382          else
383                 target_velocity =  sgn(pos_delta_rad) * joint_velocity_command_[j];
384
385          // Gazebo API function to set the joint velocity
386          sim_joints_[j]->SetParam("vel",0, target_velocity);
387         }
388
389         else {
390          sim_joints_[j]->SetParam("vel",0, 0.0);
391         }
392
393         }
394         break;
395
```

Fig. 6.15: Piece of code added to the source file "default_robot_hw.cpp"

This code is really simple. If the command position is different from the current one, the joints of the robot are commanded by a Gazebo API function that makes them turn with a target velocity until the absolute value of the position command minus the current position of a particular joint becomes less than a delta allowed error of 0.01 rad (approx. 0,573 degrees). and then the velocity is set to zero. Although not noticeable in the figure, if a command position is specified without a velocity, this will default to 1 rad/s.

Lastly, the last fragment of URDF that deserves an explanation is the inclusion of Gazebo plugins for the simulation of sensors, specifically a camera in the head, and distance sensors: one of type ultrasonic on the chest and two other rear infrared sensors inside the forearms. Nonetheless, these plugins were not written by the author of this project, but were rather provided by the master student, so little can be commented about their configuration to match the specifications of the sensors the real Raider counts with. In Fig. 6.16 is illustrated the code of the ultrasonic sensor/plugin.

Sensors in this project are not a key factor to take into account, as the focus has been primarily on the stability of the simulated Raider and the control of its several degrees of freedom, for which simulation of sensors is in principle dispensable. Even though, these sensor plugins have been incorporated into the model to leave Raider ready with as many properties as possible, just in case another person / student /developer requires making use of sensor input data to implement behaviors based on external stimulus, and to demonstrate that it is indeed fairly easy to attach sensors and other kinds of plugins for an extension of the robot capabilities.

Raider.urdf ×

```
1091  <!--Sonar plugin-->
1092  <gazebo reference="ultrasonic">
1093      <sensor type="ray" name="sonar">
1094          <pose>0 0 0 0 0 0</pose>
1095          <update_rate>5</update_rate>
1096           <visualize>true</visualize>
1097          <ray>
1098            <scan>
1099              <horizontal>
1100                <samples>5</samples>
1101                <resolution>1.0</resolution>
1102                <min_angle>-0.2617993878</min_angle>
1103                <max_angle>0.2617993878</max_angle>
1104              </horizontal>
1105              <vertical>
1106                <samples>5</samples>
1107                <resolution>1.0</resolution>
1108                <min_angle>-0.2617993878</min_angle>
1109                <max_angle>0.2617993878</max_angle>
1110              </vertical>
1111            </scan>
1112            <range>
1113              <min>0.02</min>
1114              <max>4</max>
1115              <resolution>0.003</resolution>
1116            </range>
1117          </ray>
1118          <plugin filename="libgazebo_ros_range.so" name="gazebo_ros_range">
1119            <gaussianNoise>0.005</gaussianNoise>
1120            <alwaysOn>true</alwaysOn>
1121            <updateRate>5</updateRate>
1122            <topicName>sonar</topicName>
1123            <frameName>ultrasonic</frameName>
1124            <fov>0.5</fov>
1125            <radiation>ultrasound</radiation>
1126          </plugin>
1127      </sensor>
1128  </gazebo>
```

Fig. 6.16: Ultrasonic sensor plugin in Raider.urdf

6.2.5 Control of Raider in Gazebo

In order to be able to send commands of motion to the robot, a new package from where to launch the controllers, as per ROS standards named “raider_control” had to be created within the catkin workspace files hierarchy of Section 6.2.2. The structure of this new package looks as follows:

```
../catkin_ws/src
    /raider_control      *package from where to launch the controllers*
       package.xml       *defines package properties (e.g. name)*
       CMakeLists.txt    *required for compilation purposes*
       /launch           *contains roslaunch files*
          raider_control.launch   *launches controllers*
       /config           *contains the configuration file*
          raider_config.yaml      *specifies configuration of controllers*
    /raider_description  *package for robot description files*
    /raider_gazebo       *package for robot and world simulation in Gazebo*
```

Fig. 6.17: Files hierarchy updated with the raider_control package

The content of the “raider_control.launch” file is shown in Fig 6.18. When it is invoked the first line, "rosparam", loads the controller settings to the parameter server by loading a yaml configuration file. Then, the controller_spawner node initializes a pair of position and velocity controllers for each joint. Since Raider presents 20 degrees of freedom, a total of 40 controllers are started. This is done by running a python script that makes a service call to the ros_control controller manager. The service calls tell the controller manager which controllers are requested. It also loads a “joint_states_controller” that publishes the joint states of all the joints with hardware_interfaces and advertises the topic on “/raider/joint_states”. The spawner is just a helper script for use with roslaunch.

Moreover, a “robot_state_publisher” node is started, which just listens to “raider/joint_states” messages from the “joint_state_controller” and publishes the transforms to /tf. This allows visualizing the simulated robot in Rviz for example as was already commented for figure 6.11

```
<launch>
  <rosparam file="$(find raider_control)/config/raider_control.yaml" command="load"/>
  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen">
    <remap from="/joint_states" to="/raider/joint_states" />
  </node>

  <!-- load the controllers -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" ns="/raider" args="
    middle_hip_twist_position_controller
    neck_position_controller
    l_shoulder_swing_position_controller
    l_shoulder_lateral_position_controller
    l_elbow_position_controller
    r_shoulder_swing_position_controller
    r_shoulder_lateral_position_controller
    r_elbow_position_controller
    l_hip_twist_position_controller
    l_hip_lateral_position_controller
    l_hip_swing_position_controller
    l_upper_knee_position_controller
    l_ankle_swing_position_controller
    l_ankle_lateral_position_controller
    r_hip_twist_position_controller
    r_hip_lateral_position_controller
    r_hip_swing_position_controller
    r_upper_knee_position_controller
    r_ankle_swing_position_controller
    r_ankle_lateral_position_controller
    middle_hip_twist_velocity_controller
    neck_velocity_controller
    l_shoulder_swing_velocity_controller
    l_shoulder_lateral_velocity_controller
    l_elbow_velocity_controller
    r_shoulder_swing_velocity_controller
    r_shoulder_lateral_velocity_controller
    r_elbow_velocity_controller
    l_hip_twist_velocity_controller
    l_hip_lateral_velocity_controller
    l_hip_swing_velocity_controller
    l_upper_knee_velocity_controller
    l_ankle_swing_velocity_controller
    l_ankle_lateral_velocity_controller
    r_hip_twist_velocity_controller
    r_hip_lateral_velocity_controller
    r_hip_swing_velocity_controller
    r_upper_knee_velocity_controller
    r_ankle_swing_velocity_controller
    r_ankle_lateral_velocity_controller
    joint_state_controller"/>
```

Fig. 6.18: Content of raider_control.launch

In Fig. 6.19 are showed partially the position and velocity controllers of the "raider_control.yaml" file, where the controllers name and type are listed as well the corresponding joints associated with them. For the "joint_state_publisher" a publish rate is particularly specified.

raider_control.yaml ×

```
raider:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50


  # POSITION CONTROLLERS

  neck_position_controller:
    type: position_controllers/JointPositionController
    joint: neck

  middle_hip_twist_position_controller:
    type: position_controllers/JointPositionController
    joint: middle_hip_twist

  l_shoulder_swing_position_controller:
    type: position_controllers/JointPositionController
    joint: left_shoulder_swing

  l_shoulder_lateral_position_controller:
    type: position_controllers/JointPositionController
    joint: left_shoulder_lateral

  l_elbow_position_controller:
    type: position_controllers/JointPositionController
    joint: left_elbow
```

..

..

```

  r_ankle_lateral_position_controller:
    type: position_controllers/JointPositionController
    joint: right_ankle_lateral


  # VELOCITY CONTROLLERS

  neck_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: neck

  middle_hip_twist_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: middle_hip_twist

  l_shoulder_swing_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: left_shoulder_swing

  l_shoulder_lateral_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: left_shoulder_lateral

  l_elbow_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: left_elbow

  r_shoulder_swing_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: right_shoulder_swing

  r_shoulder_lateral_velocity_controller:
    type: velocity_controllers/JointVelocityController
    joint: right_shoulder_lateral

```

Fig. 6.19: Controllers settings in the raider_control.yaml file

On the other hand, as launch files can include other launch files, becoming a nested structure in this case the “raider_world.launch” file runs everything that is necessary for simulation calling the “raider_control.launch” file, specifying the world that Gazebo should simulate, which in the figure appears as “raider_empty.world”; it also loads onto the parameter server the path where the Robot.urdf resides and determines a set of pose arguments to situate the robot from the very beginning at a desired location. It can be seen that the robot will appear in the scenario -0.85 meters in the x direction and rotated -90 degrees around the z axis. This position was simply set to make sure that Raider would appear in a world zone with direct sunlight from its front view. Moreover, a small python script “spawn_model” is employed to make a service call request to the gazebo_ros ROS node (named simply "gazebo" in the rostopic namespace) to generate the URDF-described Raider in Gazebo. This spawn_model script is located within the gazebo_ros package. It should be noted at this point that even though robots are described in URDF format to be compatible with ROS, Gazebo uses also its own format SDF, so before the simulation begins it does internally a conversion from URDF to SDF, completely transparent for the user.

*raider_world.launch ×

```
<?xml version="1.0"?>
<launch>
        <include file="$(find gazebo_ros)/launch/empty_world.launch">
                <arg name="world_name" value="$(find raider_gazebo)/worlds/raider_empty.world"/>
                <arg name="gui" value="true"/>
                <arg name="paused" value="true"/>
        </include>

<!-- Robot pose -->
  <arg name="x" default="-0.85"/>
  <arg name="y" default="0"/>
  <arg name="z" default="0"/>
  <arg name="roll" default="0"/>
  <arg name="pitch" default="0"/>
  <arg name="yaw" default="-1.5707963267949"/>

<!-- urdf xml robot description loaded on the Parameter Server-->
<param name="robot_description" textfile="$(find raider_description)/urdf/Raider.urdf" />

<!-- push robot_description to factory and spawn robot in gazebo -->
<node name="raider_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
 args="-param robot_description -urdf -x $(arg x) -y $(arg y) -z $(arg z) -R $(arg roll) -P $(arg pitch) -Y $(arg yaw) -model raider" />

<include file="$(find raider_control)/launch/raider_control.launch"/>

</launch>
```

Fig. 6.20: Content of the raider_world.launch file

Finally, if we take a look at the “raider_empty.world” file, it contains for the moment as its name suggest little to simulate: only the default sun (source of the light in the scenario) and :the default ground plane.

*raider_empty.world ×

```
<?xml version="1.0"?>
<sdf version="1.5">
<world name="myworld">

<include>
<uri>model://sun</uri>
</include>

<include>
<uri>model://ground_plane</uri>
</include>

</world>
</sdf>
```

Fig. 6.21: Content of the raider_empty.world file

Hence, now that every file involved in the simulation process has been explained, it is time to watch some action of Raider within Gazebo, which means not only graphically simulated, but also subjected to the computation of realistic physics. In Fig.6.22 Raider is displayed in the Gazebo environment after the command “roslaunch raider_gazebo raider_world.launch” was entered in a terminal window.

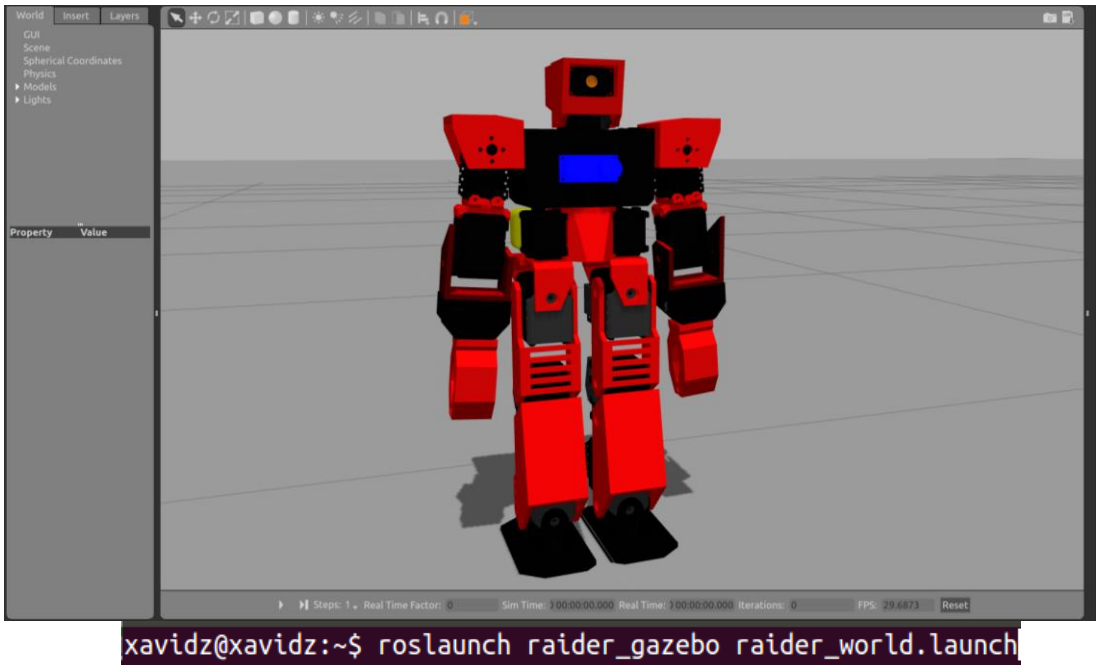


Fig. 6.22: Raider static in Gazebo

It is a good practice to check that the centers of mass are well distributed around the body. This can be done by clicking in Gazebo on View -> Center of Mass. As illustrated in Fig. 6.23, these are represented by spheres and seem symmetric for both sides.

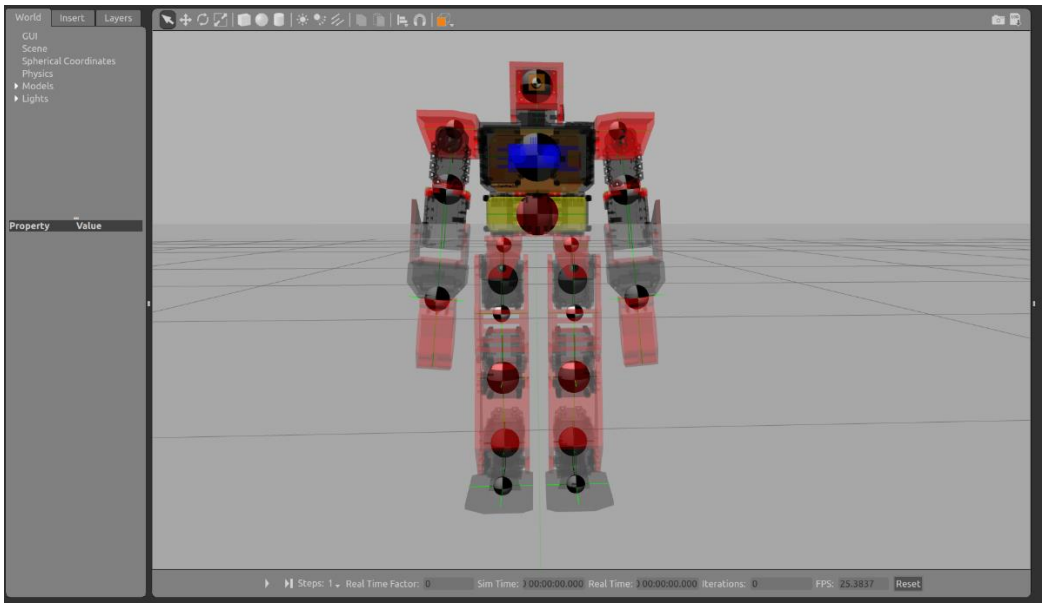


Fig. 6.23: Centers of mass of Raider displayed in Gazebo

If now the instruction “rostopic list” is executed, all topics available in the network are listed alphabetically.

```
xavidz@xavidz:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/set_link_state
/gazebo/set_model_state
/left_sharp
/raider/joint_states
/raider/l_ankle_lateral_position_controller/command
/raider/l_ankle_lateral_velocity_controller/command
/raider/l_ankle_swing_position_controller/command
/raider/l_ankle_swing_velocity_controller/command
/raider/l_elbow_position_controller/command
/raider/l_elbow_velocity_controller/command
/raider/l_hip_lateral_position_controller/command
/raider/l_hip_lateral_velocity_controller/command
/raider/l_hip_swing_position_controller/command
/raider/l_hip_swing_velocity_controller/command
/raider/l_hip_twist_position_controller/command
/raider/l_hip_twist_velocity_controller/command
/raider/l_shoulder_lateral_position_controller/command
/raider/l_shoulder_lateral_velocity_controller/command
/raider/l_shoulder_swing_position_controller/command
/raider/l_shoulder_swing_velocity_controller/command
/raider/l_upper_knee_position_controller/command
/raider/l_upper_knee_velocity_controller/command
/raider/lifecam/camera_info
/raider/lifecam/image_raw
/raider/lifecam/image_raw/compressed
/raider/lifecam/image_raw/compressed/parameter_descriptions
/raider/lifecam/image_raw/compressed/parameter_updates
/raider/lifecam/image_raw/compressedDepth
/raider/lifecam/image_raw/compressedDepth/parameter_descriptions
/raider/lifecam/image_raw/compressedDepth/parameter_updates
/raider/lifecam/image_raw/theora
/raider/lifecam/image_raw/theora/parameter_descriptions
/raider/lifecam/image_raw/theora/parameter_updates
/raider/lifecam/parameter_descriptions
/raider/lifecam/parameter_updates
/raider/middle_hip_twist_position_controller/command
/raider/middle_hip_twist_velocity_controller/command
/raider/neck_position_controller/command
/raider/neck_velocity_controller/command
/raider/r_ankle_lateral_position_controller/command
/raider/r_ankle_lateral_velocity_controller/command
/raider/r_ankle_swing_position_controller/command
/raider/r_ankle_swing_velocity_controller/command
/raider/r_elbow_position_controller/command
/raider/r_elbow_velocity_controller/command
/raider/r_hip_lateral_position_controller/command
```

Fig. 6.24: Terminal output of "rostopic list"

The topics of interest for controlling the robot are the ones ending with “/command” where the controllers wait for input commands, specifically they expect a float number that depending if the topic belongs to a position or velocity controller, it will represent the position in rad to be reached or the angular velocity in rad/s with which the motion should be done.

Additionally, one can also inspect the nodes active with “rosnode list”:

```
xavidz@xavidz:~$ rosnode list
/gazebo
/raider/controller_spawner
/robot_state_publisher
/rosout
```

Fig. 6.25: Terminal output of "rosnode list"

Or view a graph of the node connections in the ROS network:

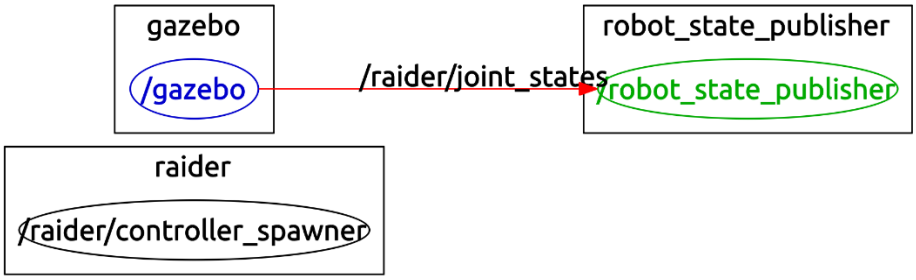


Fig. 6.26: Graph of nodes and topics relationships generated with "rosrun rqt_graph rqt_graph"

In order to send commands of movements to the robot it is possible to enter them from the terminal using the “rostopic pub” instruction like in this example where a command of 1.7 is going to be published:

```
xavidz@xavidz:~$ rostopic pub /raider/l_shoulder_swing_position_controller/command std_msgs/Float64 1.7
publishing and latching message. Press ctrl-C to terminate
```

Fig. 6.27: Example of "rostopic pub" to send a command to the "l_shoulder_swing_position_controller"

This results in the following movement executed by Raider:

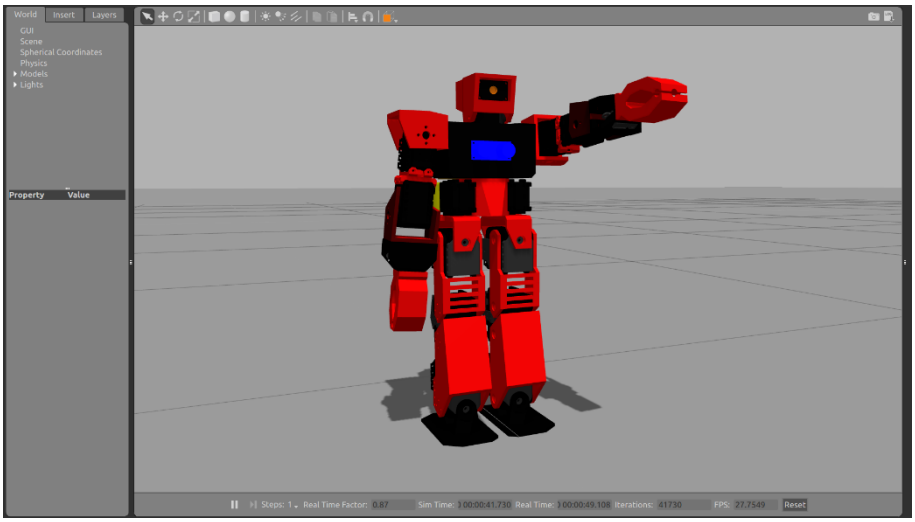


Fig. 6.28: Raider lifting its left arm after sending a command

It is also possible to retrieve some information about a certain topic with “rostopic info”, to know the message type it expects and which nodes are publishers or subscribers of that particular topic.

```
xavidz@xavidz:~$ rostopic info /raider/l_shoulder_swing_position_controller/command
Type: std_msgs/Float64

Publishers: None

Subscribers:
 * /gazebo (http://xavidz:41575/)
```

Fig. 6.29: Terminal output of "rostopic info"

However, to know which is the data being published or the last published one should use “rostopic echo”. In the figure below it is shown that the command sent to the “l_shoulder_swing_position_controler” was 1.7.

```
xavidz@xavidz:~$ rostopic echo  /raider/l_shoulder_swing_position_controller/command
data: 1.7
```

Fig. 6.30: Terminal output of "rostopic echo"

The current position of the joint can be checked in Gazebo. As we see the error between the reached position and the commanded position is minimal, less than 0,01 rad.

Property	Value
name	raider::left_shoulder_swing
type	revolute
parent link	raider::chest
child link	raider::left_shoulder
▸ pose	
angle_0	1.69000250779845
▸ axis1	

Fig. 6.31: Checking the current position in Gazebo of the joint "left_shoulder_swing"

We can try to move another joint, for example, the opposite arm to the position 2.5:

```
xavidz@xavidz:~$ rostopic pub /raider/r_shoulder_swing_position_controller/command std_msgs/Float64 2.5
publishing and latching message. Press ctrl-C to terminate
```

Fig. 6.32: Example 2 of "rostopic pub" to send a command to the “r_houlder_swing_position_controller”

The robot lifts its right arm again as ordered:

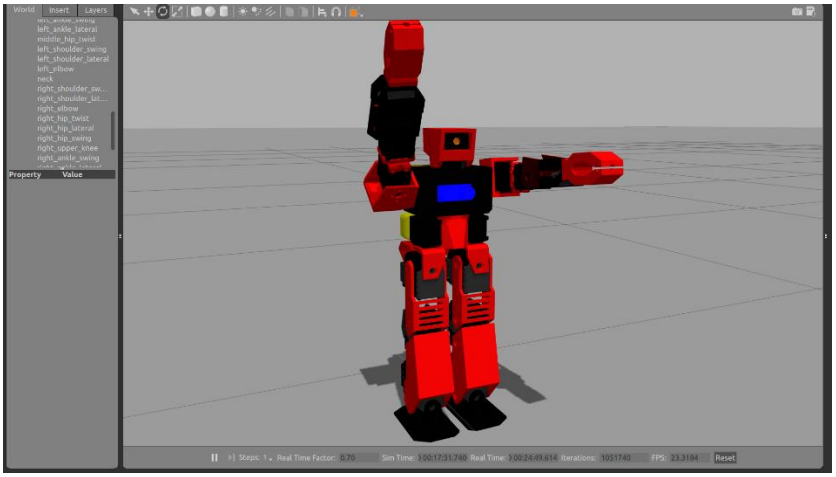


Fig. 6.33: Raider lifting its right arm after sending a second command

And the position reached by this joint can be verified once more:

Property	Value
name	raider::right_shoulder_swing
type	revolute
parent link	raider::chest
child link	raider::right_shoulder
▸ pose	
angle_0	2.49397133106441
▸ axis1	

Fig. 6.34: Checking the current position in Gazebo of the joint "right_shoulder_swing"

Finally, in the next picture it is displayed the rays of the sensors (ultrasonic and infrared) when in the URDF their visualization is set to true. As stated in at the bottom of page 68, they are not taken into account for anything is this project

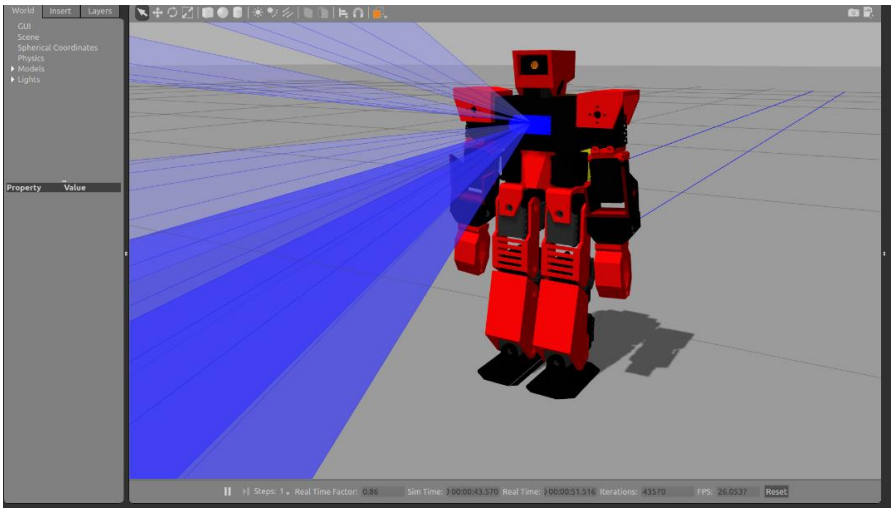


Fig. 6.35: Visualization of the rays of the Raider sensors in Gazebo

6.3 Simulation of the obstacle race event

For the creation of the 3D scenario based on the obstacle race event of the CEABOT competition, described in Chapter 3, the modeling tool Blender has been used. The scenario to be modeled consists of a bottom rectangular plane (the ground), four lateral walls and many obstacles, which are specifically six, distributed randomly by the jury before the event begins.

The measurements of the scenario, shown in Fig. 6.36, are indicated in the regulations of the competition, CEABOT 2016 and are as follows:

- Floor (2,5 x 2 meters)
- Walls (2,5 x 2 x 0,5 meters)
- Obstacles (0,2 x 0,2 x 0,5 meters)

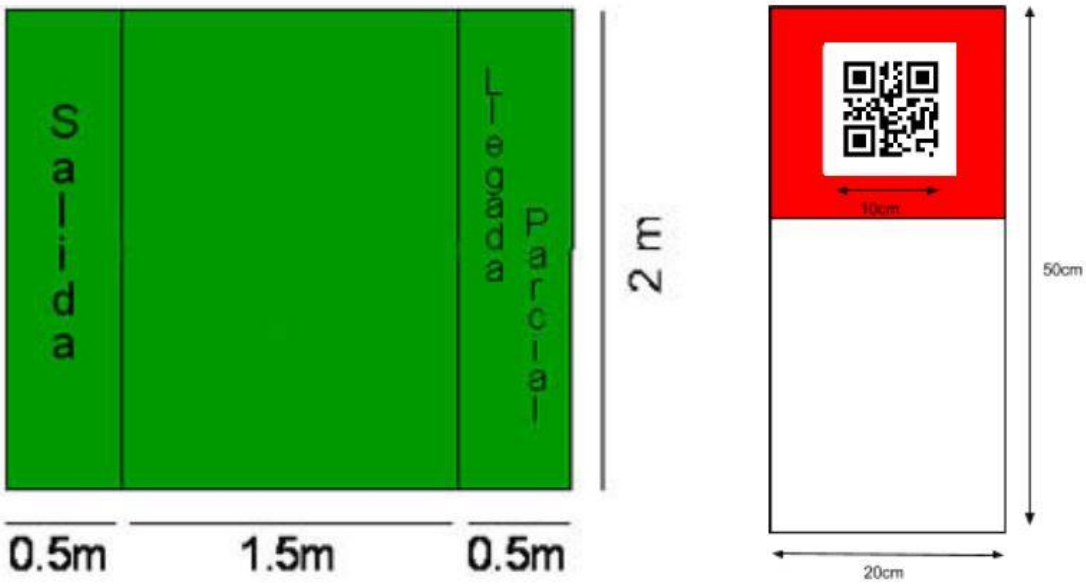


Fig. 6.36: Measurements of the obstacle race scenario

It was decided to model the components of this environment separately, so that it was easier the process of applying textures and also as result, the scenario would be much more flexible, versatile and modular. In this way, it would be easier to modify one of the pieces without having to redo the entire ensemble, so it was divided into three distinct parts: floor, walls (of two kinds) and obstacles.

6.3.1 Floor

The creation of the floor was made in turn in two stages: first a simple plane was generated in Blender with the required dimensions of 2 x 2,5m. Once done, it was proceeding with the design of a texture that could be applied to this plane, to make it look like the one of the real competition. For this task the open-source software for image editing Gimp was employed. to create a rectangle of 2000 x 25000 pixels for matching perfectly the plane generated in Blender and have a simple scale of 1pixel:1mm. Furthermore, it was colored green with the RGB values (22, 198, 6) stated in the regulation document. Also two white lines of width arbitrarily chosen (as the regulations do not specify anything about this issue), were painted onto this rectangle image to mark the start and finish zones of the scenario. The it was saved as a .png files as can be seen in Fig. 6.37

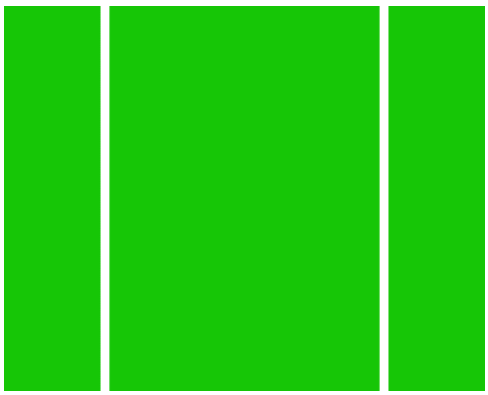


Fig. 6.37: Green rectangle with two white lines designed with Gimp

Then back in Blender a material was defined for the plane and the texture was imported. The followed method to apply the texture is called UV Mapping, which is the easiest procedure to adjust the image coordinates to the geometry in 2D. Fig. 6.38 illustrates the final look of the floor.

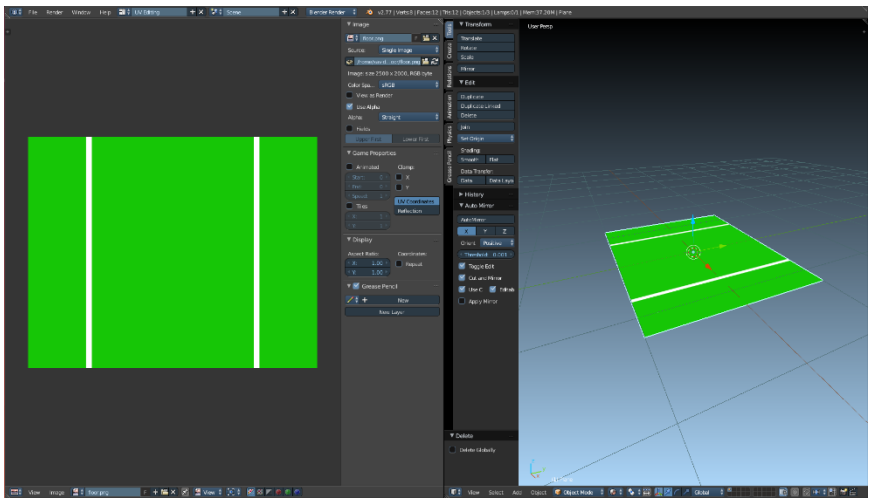


Fig. 6.38: Floor appereance after appyling the UV mapping technique

6.3.2 Walls

The wall perimeter was made out of two kinds of walls: the one of the longer rectangle side and the shorter one. The former was called wall1 and it was made as a very narrow box with the dimensions of 0.05 x 2.5 x 0.5 m. Then its outer face was separated and it was applied a texture just for decoration purposes with the UV mapping method, a process almost identical as explained for the floor. This texture corresponds to the logo and name of the university. This wall appears in Fig 6.38.

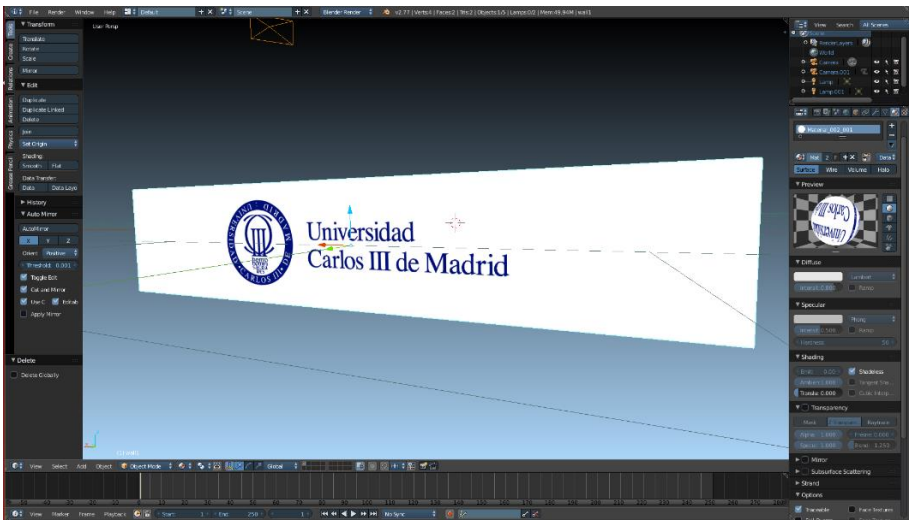


Fig. 6.39: Wall of the longer perimeter side with UC3M logo

The wall denoted as number two, corresponding to the shorter side, suffered a similar procedure; the only difference is that another texture was applied, particularly the logo of ASROB, the Robotics Association of the university as shown below. A copy of these two walls was made to have four sides that were joined and exported as single wall perimeter. The walls of this year competition have also QR codes on the inner sides, but this was not modeled as it was not too much time left for that.

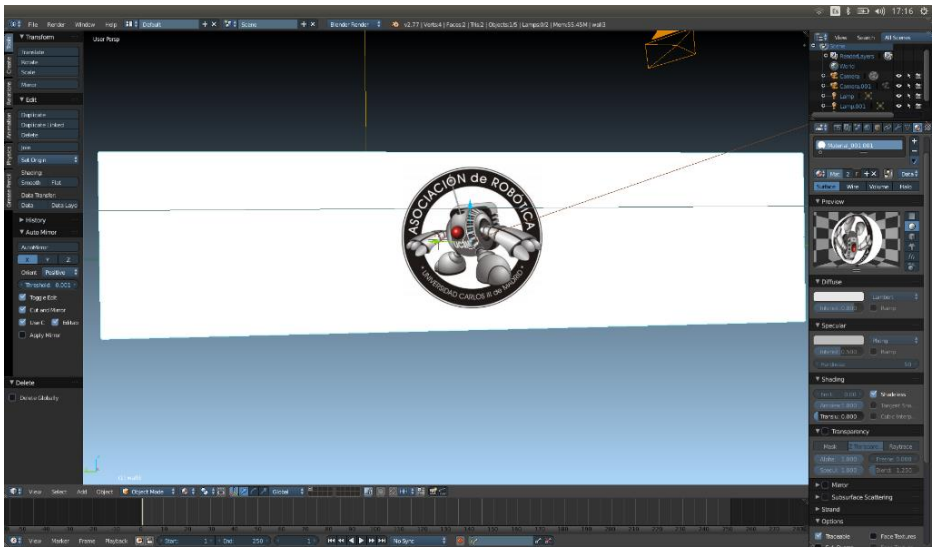


Fig. 6.40: Wall of the shorter perimeter side with the ASROB logo

6.3.3 Obstacles

Two obstacles were modeled. In previous editions of the CEABOT competition all obstacles looked exactly the same, but recently QR codes were posted on the upper parts of the obstacles to encourage the use of computer vision algorithms to solve the trials. These QR marks codify actually a simple information: the direction from which they are seen (by the participating robot). The example indicated in Fig. 6.40. means that if the robot recognizes the first QR code on the left, it should interpret that it is going north towards the first obstacle, hence the nomenclature N1. If he sees the second QR, it means it is heading east towards the first obstacle, and so on. For the second obstacle the information stored in the four QR codes would be N2, E2, S2 and W2.

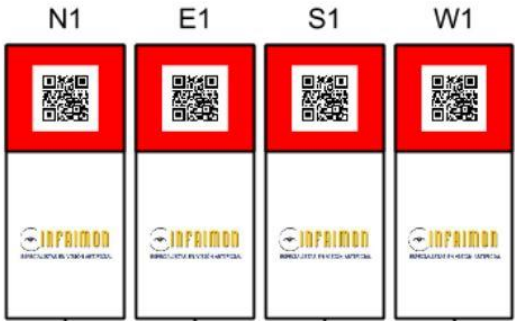


Fig. 6.41: QR codes of obstacle 1

The QR codes for both the first and second obstacles were generated as image files with the free tool “QRencoder”. Then in Gimp, they were placed over a red background of 200 x 200 pixels. At the same time in Blender a cube was generated for the lower part of the obstacles and for the portion where the QR codes should be placed, five small planes were put together: 4 for the sides and one for the top cover and finally each lateral plane (upper face of the obstacle) was applied a texture corresponding to the QR code that matches its frontal direction. The whole process was made twice, one time to get the first obstacle and the second time to get the second obstacle. The result is displayed next in Fig 6.41.

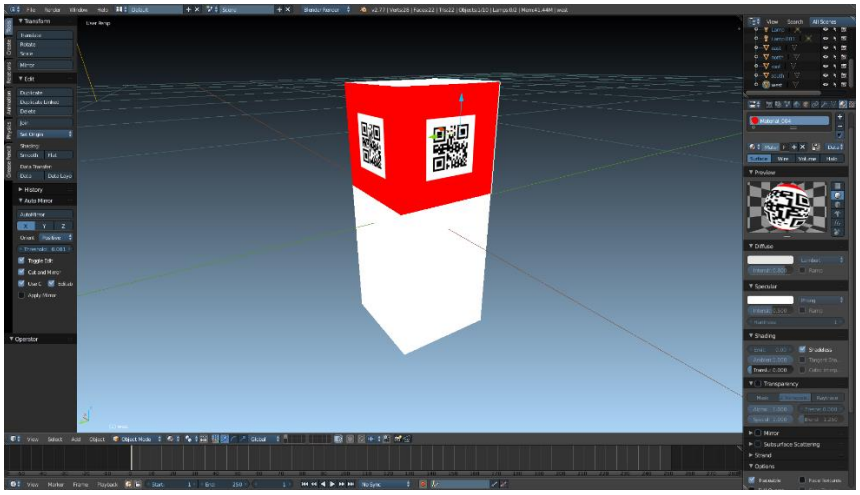


Fig. 6.42: Obstacle 1 in Blender with its four QR codes on each side

6.3.4 World file for Gazebo

The modeled pieces were exported to individual folders with self-explanatory names within the “models” directory of the “raider_gazebo” package. These models were all included in a .world file named “obstacle_race.world”. In this world file other components are specified following the specification of the SDF format supported by Gazebo. These include for instance a sky, point light sources to provide the scenario with good illumination conditions, a camera on top of the scene to watch the world from above, etc. The default sun, ground plane and origin reference frame were removed to visualize better the scenario. Also in order to open this new world, the file “raider_world.launch” was updated replacing the “raider_empty.world” by “obstacle_race.world” of which some fragments appear in the next figures:

obstacle_race.world ×

```
1 <?xml version="1.0" ?>
2 <sdf version="1.5">
3
4
5   <world name="obstacle_avoidance_contest">
6
7     <scene>
8       <sky>
9       <clouds>
10        <speed>12</speed>
11      </clouds>
12     </sky>
13     <grid>0</grid>
14     <origin_visual>0</origin_visual>
15    </scene>
16
17    <!-- Point light sources -->
18
19    <light name='point_light_1' type='point'>
20      <pose frame=''>0 3 2 0 0 0</pose>
21      <diffuse>0.498039 0.498039 0.498039 1</diffuse>
22      <specular>0.0980392 0.0980392 0.0980392 1</specular>
23      <attenuation>
24        <range>20</range>
25        <constant>0.5</constant>
26        <linear>0.01</linear>
27        <quadratic>0.001</quadratic>
28      </attenuation>
29      <cast_shadows>0</cast_shadows>
30      <spot>
31        <inner_angle>0.6</inner_angle>
32        <outer_angle>1</outer_angle>
33        <falloff>1</falloff>
34      </spot>
35      <direction>0 0 -1</direction>
36    </light>
37
38    <light name='point_light_2' type='point'>
39      <pose frame=''>0 -3 2 0 0 0</pose>
40      <diffuse>0.494118 0.494118 0.494118 1</diffuse>
41      <specular>0.0941176 0.0941176 0.0941176 1</specular>
42      <attenuation>
43        <range>20</range>
44        <constant>0.5</constant>
45        <linear>0.01</linear>
46        <quadratic>0.001</quadratic>
```

Fig. 6.43: Fragment 1 of the obstacle_race.world file

```
94
95    <model name="floor">
96      <pose>0 0 0 0 0 0</pose>
97      <static>true</static>
98      <link name="plane">
99        <visual name="visual">
100         <geometry>
101           <mesh><uri>model://floor/floor.dae</uri></mesh>
102         </geometry>
103       </visual>
104       <collision name="collision">
105         <geometry>
106           <mesh><uri>model://floor/floor.stl</uri></mesh>
107         </geometry>
108         <surface>
109           <friction>
110             <ode>
111               <mu>100</mu>
112               <mu2>50</mu2>
113             </ode>
114           </friction>
115           <contact/>
116         </surface>
117         <max_contacts>10</max_contacts>
118       </collision>
119     </link>
120   </model>
121
122   <model name="walls">
123     <pose>0 0 0.25 0 0 0</pose>
124     <static>true</static>
125     <link name="perimeter">
126       <visual name="visual">
127         <geometry>
128           <mesh><uri>model://walls/walls.dae</uri></mesh>
129         </geometry>
130       </visual>
131       <collision name="collision">
132         <geometry>
133           <mesh><uri>model://walls/walls.stl</uri></mesh>
134         </geometry>
135       </collision>
136     </link>
137   </model>
```

Fig. 6.44: Fragment 2 of the obstacle_race.world file

And finally some pictures of the simulated scenario with Raider in Gazebo:

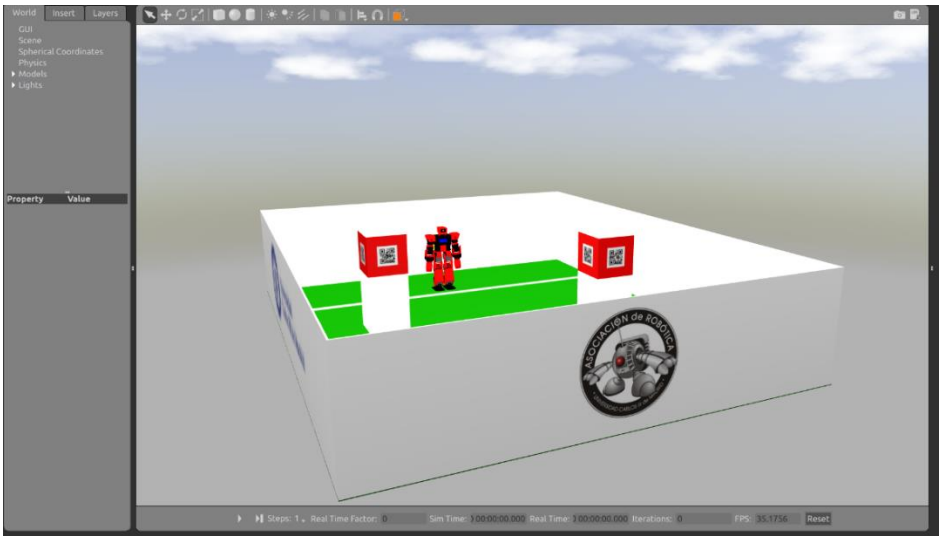


Fig. 6.45: Obstacle race world - perspective 1

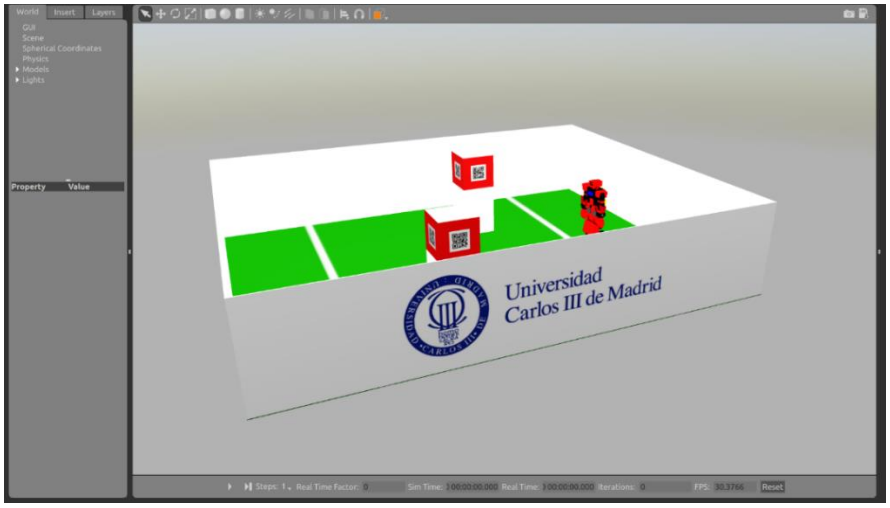


Fig. 6.46: Obstacle race world - perspective 2

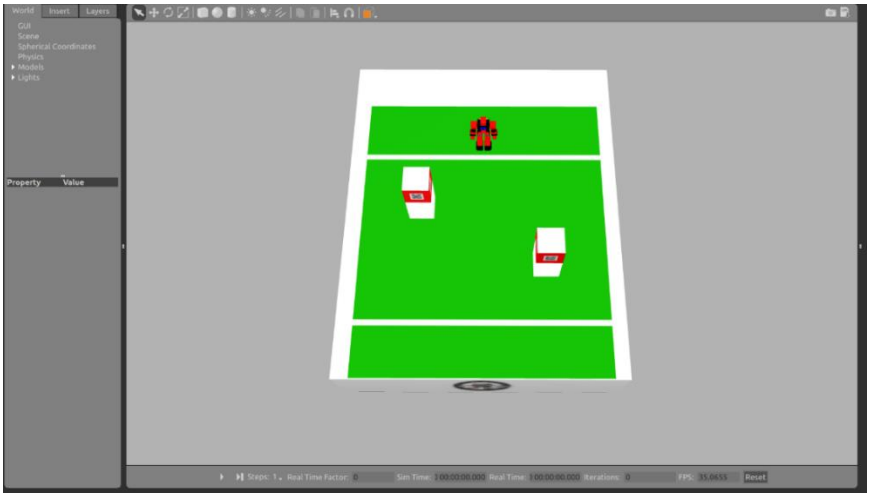


Fig. 6.47: Obstacle race world - perpective 3

6.4 Design of a locomotion controller

The control of Raider has been tested in Section 6.2.5 only through input commands from a terminal. This is fine for just testing the motion of the individual joints, however, it is definitely not a good approach to simulate more complex movements that involve the coordination of multiple joints at the same time.

For doing this, it is much better to build some kind of controller that can publish many joint values at once. As outlined in the introduction of this chapter, the software chosen for designing a controller was MATLAB /Simulink, mainly because of the recent native support for ROS through the Robotics System Toolbox add-on. But in order to achieve the implementation of a “sophisticated” controller, first the control of each joint of Raider was tested in the Simulink environment.

```
xavidz@xavidz:~$ rosnode list
/Raider_walk_81473
/gazebo
/raider/controller_spawner
/robot_state_publisher
/rosout
```

Fig. 6.48: Terminal output of "rosnode list" listing a node "Raider_walk...", which is the name of a Simulink model

The Robotics System Toolbox lets Simulink interface with ROS by creating a node out of the model that can share information in the way like any other regular node does. In Fig 6.48 it is showed that if we hit “rosnode list” in the terminal, there is listed a node with the name of the Simulink model “Raider_walk” followed by some digits. When the model is run, it connects by default to the ROS master in the network identified by the localhost URI, but it is possible to change this setting and enter another IP direction to permit communication between different network devices. The toolbox provides a library with three specific ROS blocks that can be dragged into a Simulink model: a Blank Message, a Publisher and a Subscriber block. These were used in a model like the one depicted in Fig. 6.49.

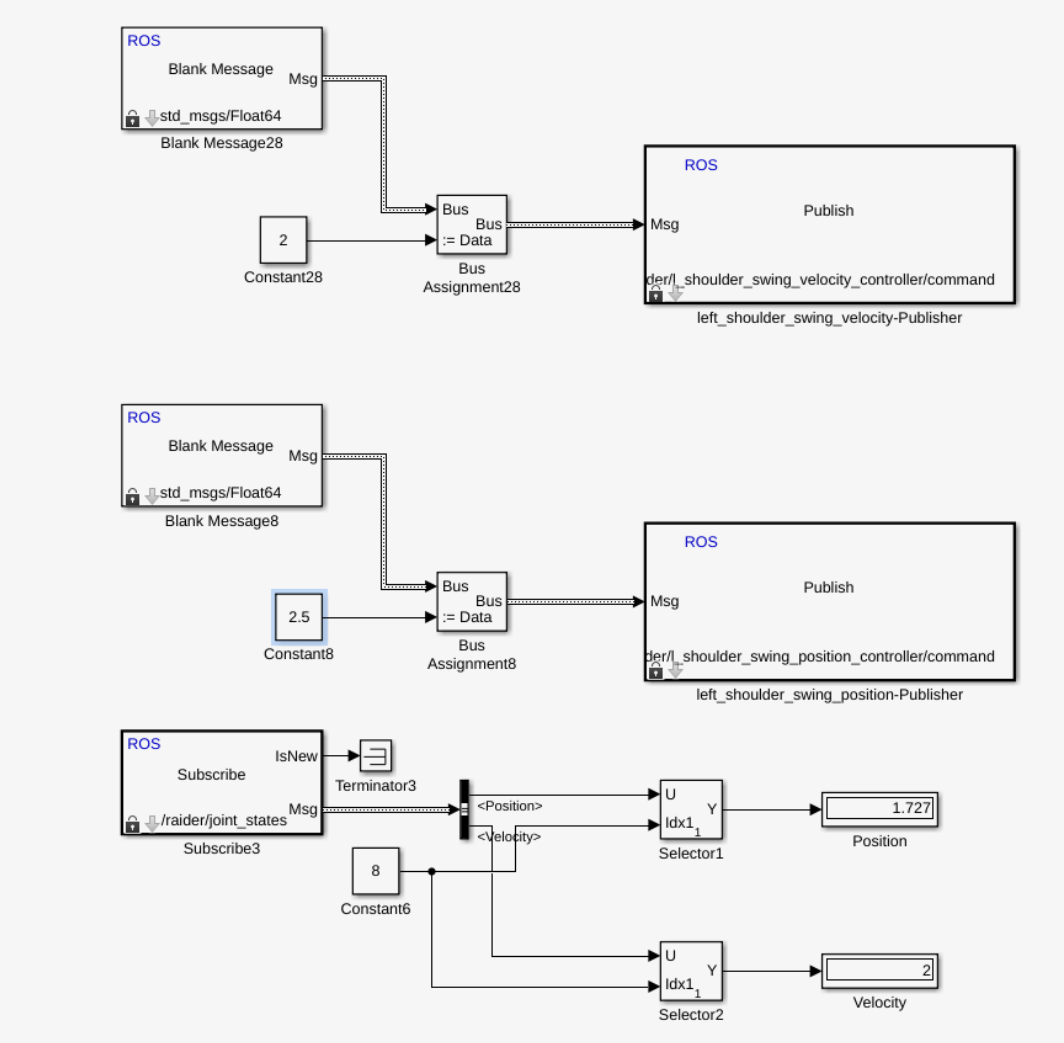


Fig. 6.49: Initial testing of joint control in Simulink

In the previous picture, a simulation of Raider was previously started with “roslaunch raider_gazebo raider_world”. In that model, the Blank Message blocks expect to be filled with the ROS message data type they support (std_msgs/Float64). Because the message is considered in Simulink as a bus signal, they are connected to Bus Assignment blocks where the value of the Constant blocks is assigned. Moreover, there are two Publisher blocks related to the same joint, specifically to the “left_shoulder_swing”, but one of them publishes to the “…position_controller/command” topic, hence the name “…position-Publisher” and the other does the same, but to the “…velocity_controler/command”, hence the name “…velocity-Publisher”.

Below them, there is a Subscriber block that listens to the topic “/raider/joint_states”. Under this topic (whose publisher in the ROS network is the “joint_state_controller”) the current variables that define the state of joints are published altogether as a set of position, velocity, and effort arrays, thus the Bus Selector block the Subscriber is connected to, is needed to output these arrays further individually in the model. Then, as these buses transport respectively position and velocity arrays containing the current values of all joints, they are in turn linked to other Selector blocks to extract only the data related to a particular joint. In the example, the array element that corresponds to the joint “left_shoulder_swing” is in the eighth position of the array (joints are stored in the arrays in alphabetical order), so that’s why an “8” is the input for the Selector blocks.

In the moment the screenshot was taken, it was being published a command of 2.5 rad in the position topic and a command of 2 rad/s in the velocity topic. And as it is shown in the Velocity Display block, that velocity is being reached without problems while the joint is moving to the target position (at the instant the screenshot was taken, the joint was still moving towards the commanded position, therefore in the Position Display block appears an intermediate angle of 1.272 rad,).

Now, Fig. 6.50 is another snapshot of the same model approx. half a second later. In this occasion the Position Display shows 2.497 rad, almost the commanded 2.5 rad, and the Velocity Display shows 9.74e-05, so practically 0 rad/s. What both screenshots of the model are in summary confirming is that the joint has moved with the specified desired velocity to reach in a short amount of time the commanded position and then it stopped, exactly what it was supposed to do. That’s the most important conclusion!

These control behavior was verified with all joints in a Simulink model called “Raider_test.slx” and the precision and results were almost identical, which was the starting point for thinking about a more sophisticated control algorithm.

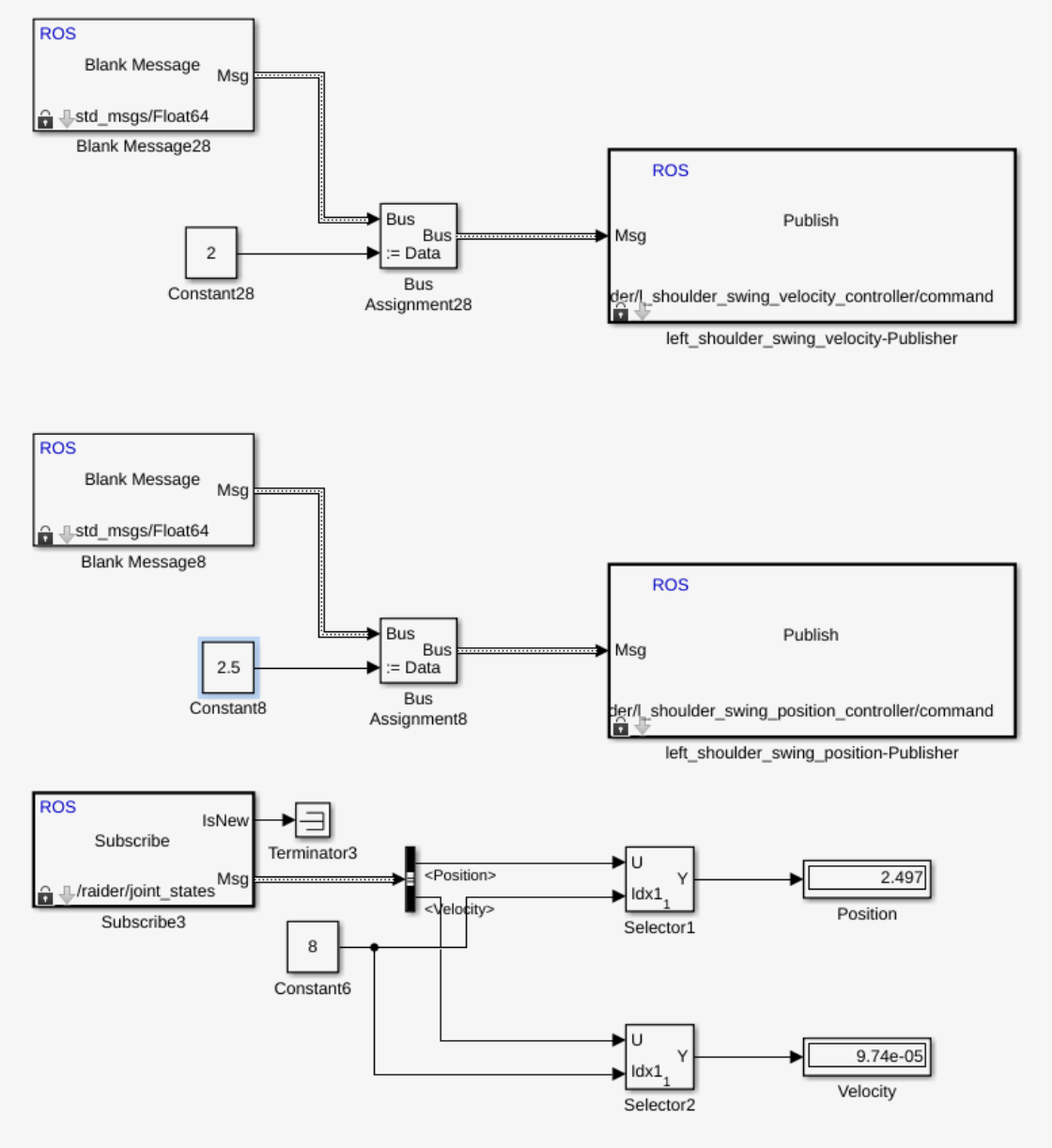


Fig. 6.50: Same Simulink model as in the previous figure, only approx. half a second later

6.4.1 Raider_walk

"Raider_walk" is the name of the controller designed to try to reproduce the walking gait of the robot with similar pattern and functions used on the real Raider. Its overview can be seen in Fig. 6.51. A C++ library of movements for Raider was developed by Javier Isabel, the original creator of the real robot. The implementation explained next is mainly based on his file "raider_motion.cpp" [40], but adapted to the Simulink environment and Matlab language.

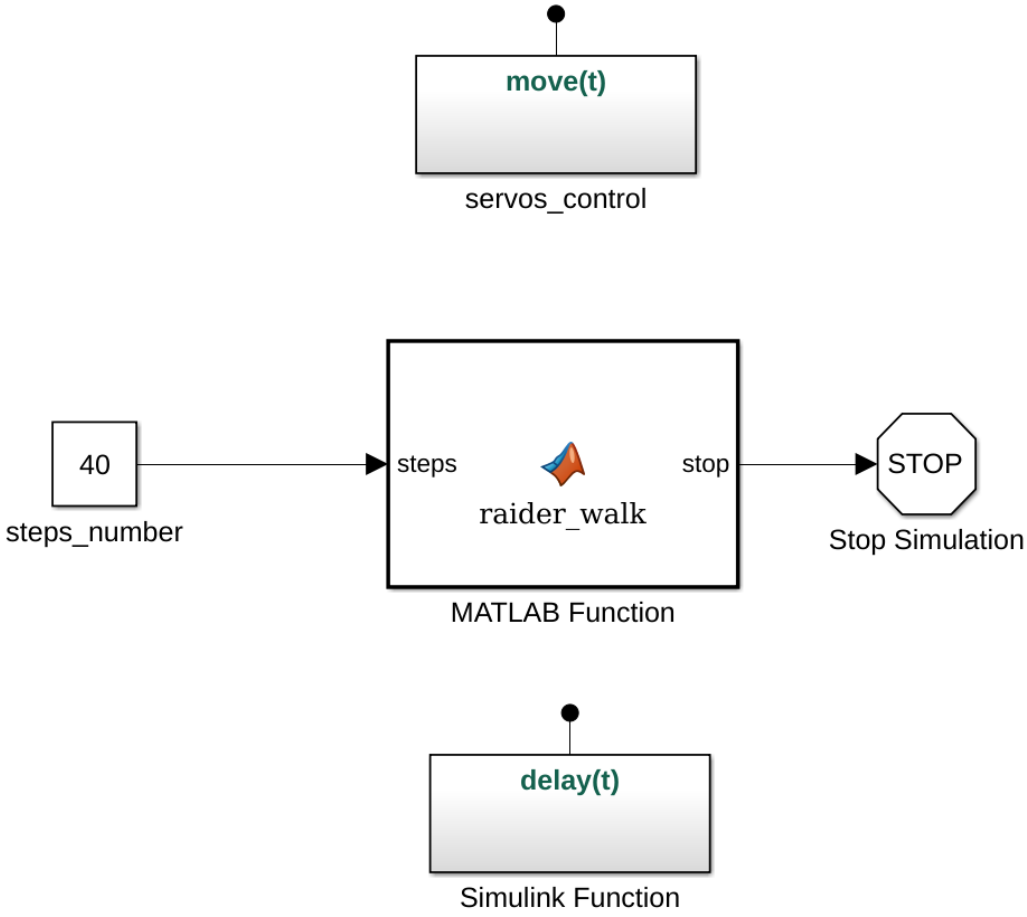


Fig. 6.51: Raider_walk controller overview

In Simulink, the "Raider_walk.slx" model relies on this elements:

- function **raider_walk(steps):** is the core of the controller that contains that contains the walk algorithm. It takes as input the number of steps the robot should walk and stops the simulation when it finishes.
- function **publishPosition(joint, pos):** takes as input a certain position "pos" that is routed to the corresponding joint_position-Publisher block to be published.
- function **publishVelocity(joint, vel):** take as input a certain velocity "vel" that is routed to the corresponding joint_velocity-Publisher block to be published.
- **currentPosition(20):** an array that stores the current saved positions of all joints.
- **targetPosition(20):** an array that stores target positions for all joints.
- funtion **setTargetPosition(20):** has access to the targetPosition(20) array to assign new values to its elements.
- function **updateCurrentPosition(20):** copies the targetPosition(20) array values into the currentPosition(20) array when a motion has finished.
- function **move(t):** probably the most important of all since it commands the joints. It takes as input the time in seconds (typical values are between 0.5 and

0.1 seconds) in which the joints should reach new postions. It calls the publish_Position function in a loop with all the values of the targetPosition(20) array. Also, it computes the absolute value of the difference between the currentPosition(20) and targetPosition(20) for all joints and divides it into the time specified to send different velocity commands to each joint calling the publishVelocity function, so that they reach positions at the same time and make simultaneous motions and the inertial disturbance is lower.

- function **delay(t):** also a quite essential function of the model. It is used to pause "t" seconds (the time specified in move(t)) the execution of the program when a movement is being done by the robot. Additionally, it is used to delay the walking 40 ms before each step, as it was realized that the simulated Raider was more stable with this small delay between steps.
- function **moveVertical(left_cmd, right_cmd):** uses setTargetPosition to specify positions about the "hip_swing", "upper_knee" and "ankle_swing" joints of either the left or right legs or both to achieve a vertical movement of the leg.
- function **moveForward(left_cmd, right_cmd):** uses setTargetPosition to specify positions about the "hip_swing", and "ankle_swing" joints of either the left or right legs or both to achieve a forward movement of the leg, although with negative values it could also be used to move the leg backwards.
- function **moveShoulderSwing(left_cmd right_cmd):** uses setTargetPosition to specify positions about the "shoulder_swing" joints of either the left or right shoulders or both to achieve a forward/backwards movement of the shoulders.

An example of how moveVertical and moveForward manipulate the robot has been reproduced in Rviz and is illustrated in Fig. 6.52.

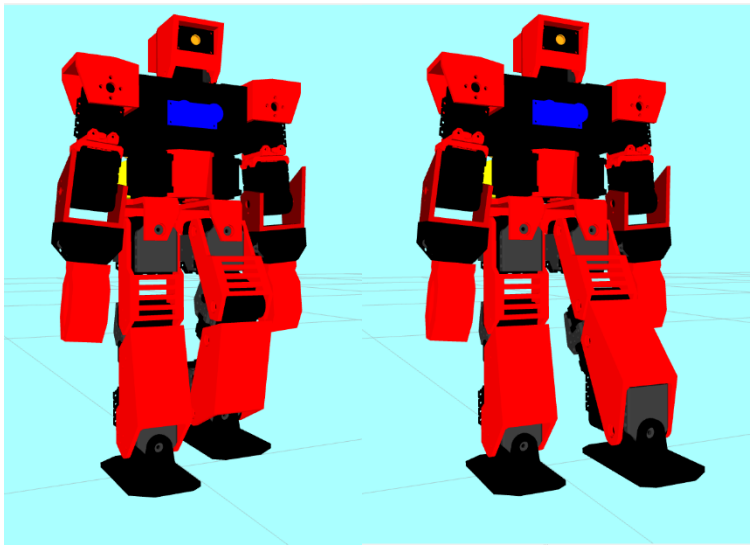


Fig. 6.52: A random example of moveVertical (left) and moveForward (right) motions

Some of the functions described were written as regular Matlab functions, which by the way must be located in the same directory of the model, or Matlab blocks, which contain also regular Matlab code, but are treated in runtime as blocks; and others were implemented as Simulink functions. Next, the last ones will be described a bit more.

6.4.1.1 move(t)

Fig. 6.53 displays the internal composition of this Simulink function. It includes a Matlab block called "move_joints" where the calculation of the particular velocity for each joint is computed as described in the previous page and both the target positions and velocity commands are passed to the publishPosition and PublishVelocity functions, which are also Simulink functions.

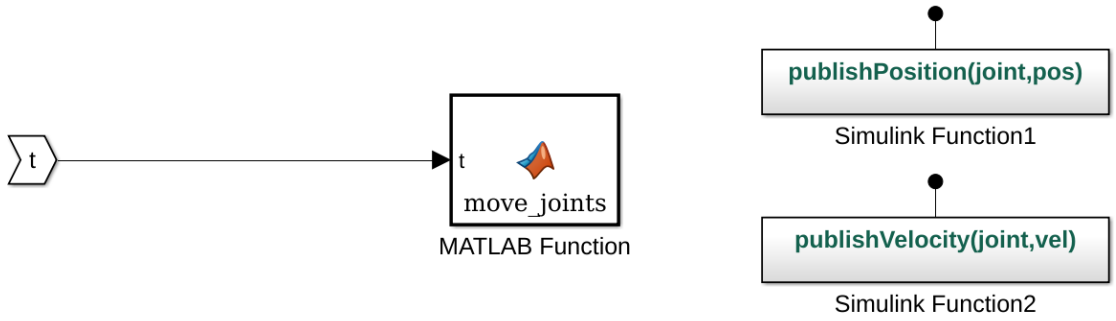


Fig. 6.53: Internal composition of the move(t) Simulink function.

The inside of the publishPosition(joint, pos) Simulink function looks as follows:

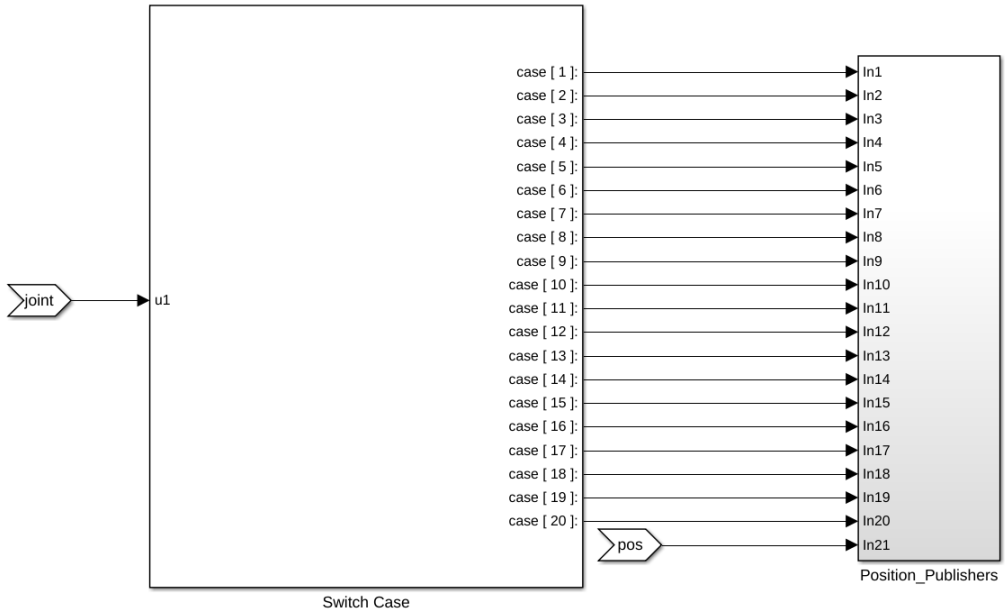


Fig. 6.54: Inside of the publishPosition Simulink function

It consists of a switch block whose input variable is "joint" and a block of Position_Publishers, where the joint_position-Publisher blocks (like the one presented in Fig. 6.49 or 6.50) for all joints are collected. Depending on the "joint" variable, which is a number from 1 to 20 representing the elements of the joints array, the data contained in the "pos" input (a position command in rad) is routed to the corresponding joint_position-Publisher to be published to the topic of the corresponding position_controller for that joint. This explanation is also applicable to the publishVelocity Simulink function, but of course in that case a velocity command in rad/s is being sent to the Velocity_Publishers block.

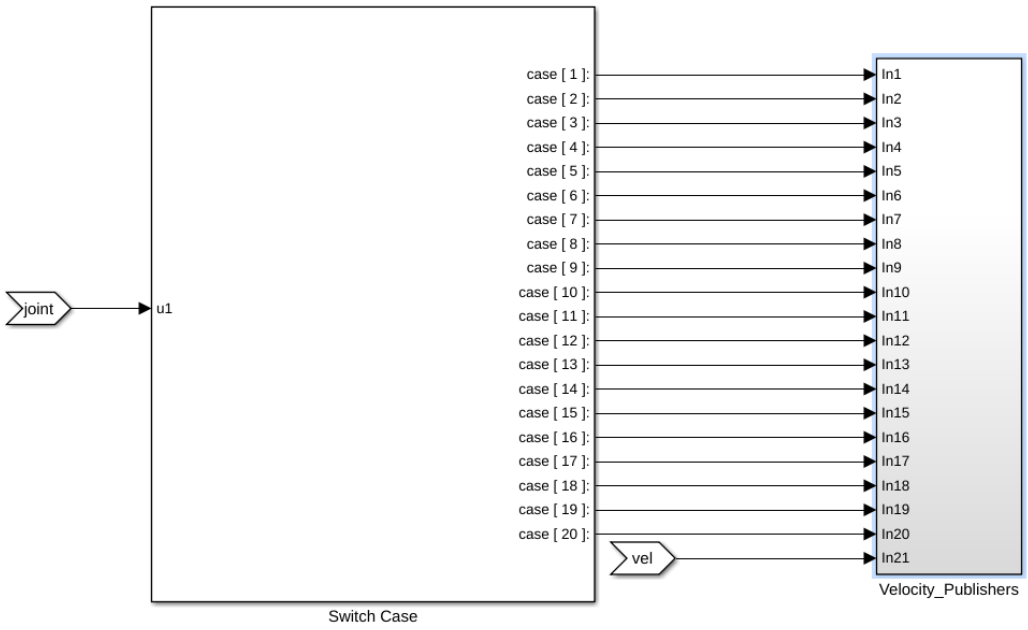


Fig. 6.55: Inside of the publishVelocity Simulink function

6.4.1.2 delay

In Fig. 6.56 we see that this function is composed of a Matlab block called "delay" as well and of another Simulink function called "time = getSimTime()". The Matlab block receives as input the desired time to delay the execution of the controller model, then it calls getSimTime to know the current simulation time and stores this value. Finally, in a while-loop it compares if the elapsed time in simulation matches the desired delay time, subtracting the stored value from the time returned by calling again getSimTime.

Since the simulation time is affected by the real-time factor, which defines how much real time passes with each simulation step and this is constantly varying, the elapsed time in Gazebo never coincides with the elapsed time in Simulink. The Robotics System Toolbox does not provide currently any automatic solution to this issue, so the delay function was implemented as a workaround to synchronize Simulink and Gazebo.

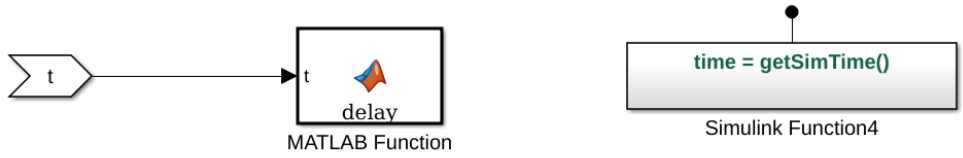


Fig. 6.56: Internal composition of the delay Simulink function

The inside of the getSimTime Simulink function looks as follows

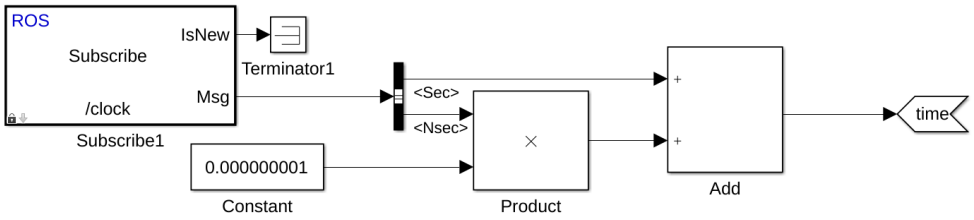


Fig. 6.57: Inside of the getSimTime Simulink function

There is a Subscriber block that listens to the topic "/clock" where Gazebo is publishing the current simulation time. However, the time format in ROS is given separately as seconds and nanoseconds. That's why the time value of the signal "Nsec" is first converted to seconds and then added to the value of the signal "Sec". The result is then returned by the function.

6.4.2 Gait pattern

Basically, the pattern to achieve the bipedal locomotion of the robot is a successive combination of the moveVertical, moveForward and moveShoulderSwing functions. Each of them sets a target position increment in radians for a few joints, so the position arguments passed to this functions (α, β and γ respectively) must be always with respect to the previous motion. Tables 6-1 and 6-2 indicate the name of the joints of Raider and its order within the targetPosition(20) array.

Joint name	Order in targetPosition(20)
middle_hip_twist	1
neck	2
right_shoulder_swing	3
left_shoulder_swing	4
right_shoulder_lateral	5
left_shoulder_lateral	6
right_elbow	7
left_elbow	8

Table 6-1: Order in targetPosition(20) of the joints of Raider

Joint name	Order in targetPosition(20)
right_hip_twist	9
left_hip_twist	10
right_hip_lateral	11
left_hip_lateral	12
right_hip_swing	13
left_hip_swing	14
right_upper_knee	15
left_upper_knee	16
right_ankle_swing	17
left_ankle_swing	18
right_ankle_lateral	19
left_ankle_lateral	20

Table 6-2: Order in targetPosition(20) of the joints of Raider (cont.)

These particular order corresponds to the IDs assigned to the Dynamixel AX-12A servos of the real robot. Below, in Table 6-2, the incremental positions of the joints involved in the gait cycle are listed for 3 steps, then the pattern repeats itself from step 2. The numbers under the "Joint" column are the ones of the previous table. During the initial implementation many different values for α, β and γ were tested and finally it was found that the best parameters to have a stable gait in simulation are:

$$\alpha = 0.2 \quad \beta = 0.1 \quad \gamma = 0.15$$

Furthermore, the robot at start is standing; all joints are in zero radians in this pose. The robot prepares itself for walking bending its legs slightly to lower the global center of mass. This movement is labeled in Table 6-3 as "0". After that is when the robot really begins to walk until the specified number of steps is reached. Each step is executed in 2 movements of 0.1 seconds each. These are indicated in the table as 1st, 2nd.

Joint	**Steps**						
	0	1		2		3	
	-	1st	2nd	1st	2nd	1st	2nd
3	0	γ	0	-2γ	0	2γ	0
4	0	$-\gamma$	0	2γ	0	-2γ	0
13	+0.55	$\alpha+\beta$	$-\alpha$	$-\beta$	0	$\alpha+2\beta$	$-\alpha$
14	-0.96	$-\beta$	0	$\alpha+2\beta$	$-\alpha$	$-\beta$	0
15	+0.55	-1.75α	$+1.75\alpha$	0	0	-1.75α	$+1.75\alpha$
16	-0.96	0	0	-1.75α	$+1.75\alpha$	0	0
17	+0.41	$0.75\alpha-\beta$	$-0,75\alpha$	β	0	$0.75\alpha-2\beta$	$-0,75\alpha$
18	+0.41	β	0	$0.75\alpha-2\beta$	$-0,75\alpha$	β	0

Table 6-3: Increment positions of the gait cycle for 3 steps

So, finally some pictures of the first steps of Raider in the obstacle race world. There is also an uploaded video of Raider walking in the scenario that can be seen in [41]

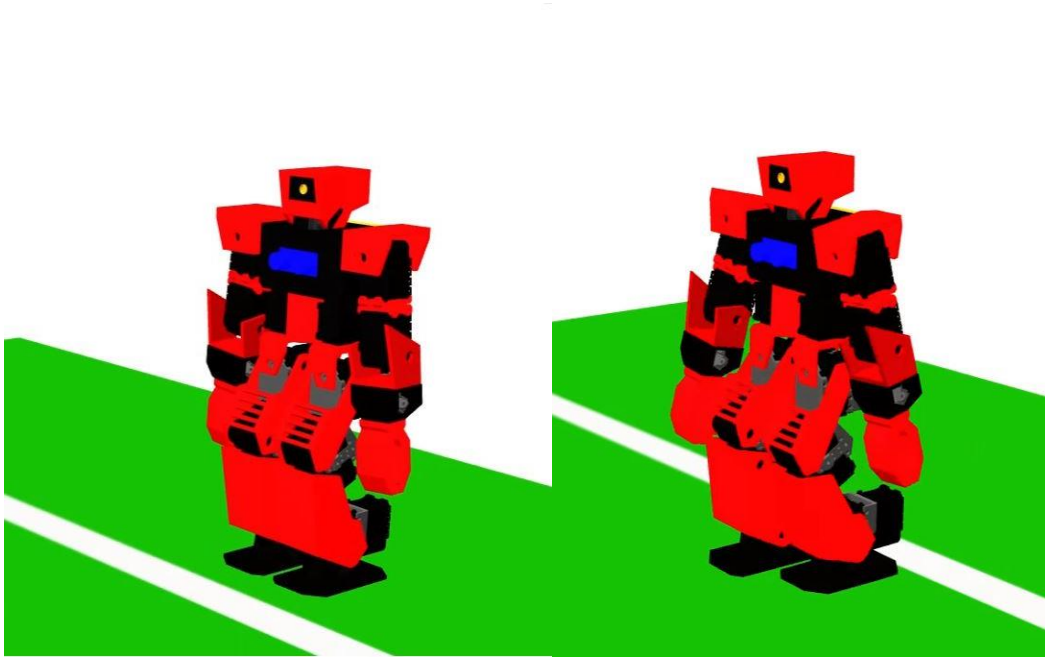


Fig. 6.58: Raider bending its legs (left) and walking (right):

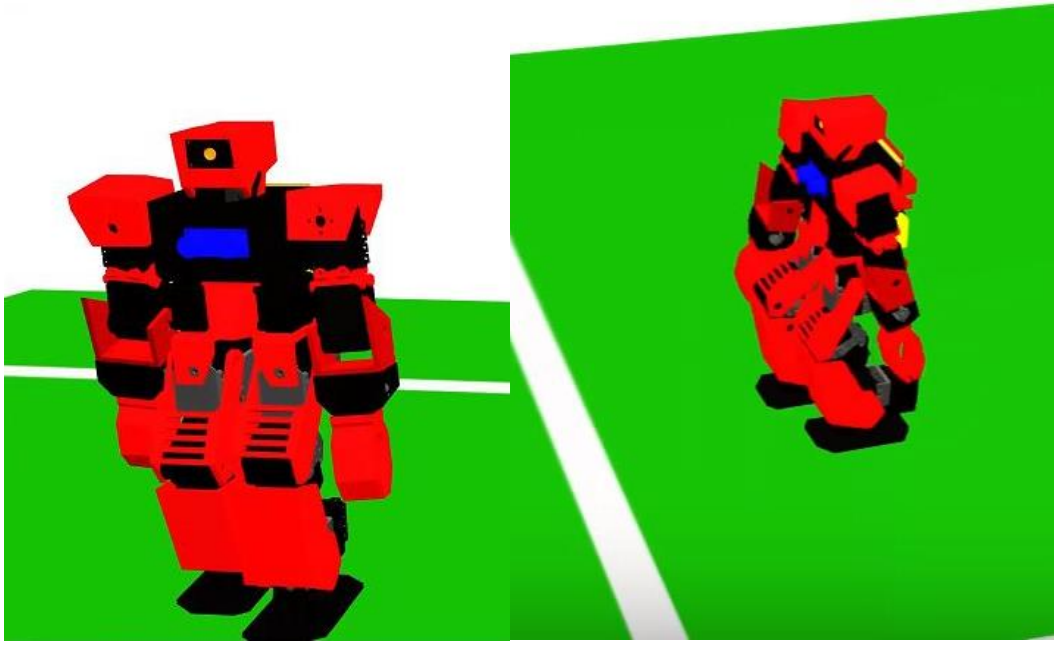


Fig. 6.59: Raider walking from different perspectives

Chapter 7

Budget

In this chapter the necessary budget for the development of this project is detailed. It will be taken into account both the salary of the workers as well as the costs derived from equipment, program licenses, etc.

In the first place, regarding the quantity of hours devoted to this project, a good estimation would be 4 hours a day during 90 days, thus in total 360 hours. Only one engineer was employed for the entire elaboration process, so this engineer has worked 360 hours and the advisor 36 hours (approximately 10% of the time). The cost per hour of the workers and the total costs are illustrated in Table 7-1

Employees	Working time (hours)	Cost per hour (€/h)	Cost (€)
Advisor	36	80	2.880
Engineer	360	40	14.400
		TOTAL	**17.280 €**

Table 7-1: Costs of personnel

Secondly, what concerns the equipment used to carry out the project, this includes a laptop and the license of the MATLAB software [25]. The rest of the programs were all of them open-source, therefore they do not contribute to the budget.

In order to determine the costs of these resources, the depreciation produced over the work period has to be computed. This has been done applying the percentage of loss in value established by the Spanish Tax Agency [42]. The formula to calculate it is the following:

$$Cost = Coeff.of\ depreciation \times \frac{N°\ of\ usage\ months}{12} \times Price\ of\ resource$$

Next, Table 7-2 indicates the individual equipment costs and the total costs of this category:

Equipment	Unitary price (€)	Nº of usage months	Coeff. of depreciation	Cost(€)
1 x Laptop	950	6	26%	123,5
1 x MATLAB license	163	5	26%	21.19
			TOTAL	**144,69 €**

Table 7-2: Equipment costs

Now in order to calculate the total cost of the project, the personnel costs, the indirect costs of equipment (that are 20% of the direct costs) and the VAT (21%) will be added. This is shown in Table 7-3.

	Cost (€)
Costs of personnel	17.280
Costs of equipment	144,69
Indirect costs (20%)	28,94
Subtotal	17.453,63
+ VAT (21%)	3.665,26
TOTAL	**21.118,89 €**

Table 7-3: Total costs

Consequently, the total cost of the project amounts to 21.118,89 €.

Chapter 8

Conclusions and future work

For finishing and summarizing the work detailed in this document, the general conclusions of the project will be presented in this chapter as well as the possible improvements that could be carried out in the future.

8.1 Project conclusions

The project started with an introduction to the science of humanoid robotics, which gave an overview of its main characteristics. It specially stands out among them that the best up-to-date technology found today is constantly applied, thus this makes it definitely an exciting research area for students around the world. In fact, the major achievements in this field were also commented, specifically it was talked about the astonishing ASIMO robot produced by Honda, the best humanoid ever made and a reference for the industry.

After this brief opening, the resources available within the Robotics Association of the Universidad Carlos III were analyzed, focusing on mini-humanoid platforms such as the Bioloid and Robonova commercial kits. Then, the main hardware features and aspects that were essential in the construction of Raider, the robot used for simulation in this project, was also described. Next, CEABOT, the Spanish competition for mini-humanoid robots was outlined, since on one hand the participating students representing the university obtained very good results in various editions and on the other hand, the subsequent modeling of a scenario was going to be inspired by the event known within CEABOT as obstacle avoidance race.

Once the sections previously mentioned were covered, it was time to investigate the several alternatives offered by the market to create development environments for mini-humanoid robots; this let us know which were its strengths and weaknesses, so that it could be much easier to choose afterwards the most convenient options for this project. The current tendency of software development, namely the component-based modelling technique was analyzed citing the concept of robotics middleware as a perfect example of this approach that encourages code reuse and enables a simple coupling of different systems with little effort.

With more information and background thanks to the study of the state of the art, it was straightforward the selection of software tools that could help implement the development environment respecting the objectives, which were a robust but modular and flexible design to be able to modify or replace it if eventually necessary. These selected software components were then reviewed remarking its useful functionalities in relation with the work assigned.

At this point, the project elaboration process could begin dividing it into three main different components. The first one was the tremendous popular ROS communications middleware that acts as a bridge connecting the other two independent applications. In order to take advantage of its many capabilities, first a virtual model of Raider to be used in simulation was built in the required URDF format. In the second place, Gazebo, a powerful and sophisticated robotics simulator that can recreate realistic physics conditions and perfectly integrated into the ROS ecosystem, was employed to observe and check the dynamic behavior of the simulated robot. Furthermore, an initial and primitive manipulation of Raider could be tested through the control infrastructure provided by the ros_control package. Lastly, in Simulink, which is an ideal piece of software for intuitive and graphical programming, a bipedal locomotion controller that could make the robot execute a walking gait, was successfully designed based on ROS specific diagram blocks provided recently by the Robotics System Toolbox add-on. The union of these three components has led in the end to the creation of a development environment suitable for further experimenting of algorithms.

8.2 Future improvements

The development environment should be enhanced in terms of simulation in such a way that the virtual Raider can compete in the trials of CEABOT with satisfactory results. In order to achieve that goal, first the scenarios left (sumo wrestling and crossover stairs) need to be modelled with accuracy. Moreover, apart from the simulated camera, and ultrasonic and infrared sensors, other plugins to instantiate for example a compass and an accelerometer could be incorporated to the URDF file. Regarding the dynamics of Raider, perhaps it is likely to achieve a better tuning of the inertial values.

On the control side, apart from walking, other complex motions of the real Raider could be implemented like jump, run, get up, turn, kick, punch, lateral displacement, etc. Also behaviors that rely on sensor data and a computer vision functionality with the OpenCV and ZBar libraries would be fantastic to deal with collision detection, path finding and recognition of QR codes respectively. To conclude, the very useful ROS packages for motion planning, navigation and localization could be tested to increase considerably the robot autonomy.

References

[1] A. Roberts, The History of Science Fiction, New York: PALGRAVE MACMILLAN, 2006.

[2] I. Asimov, I, Robot, Gnome Press, 1950.

[3] American Honda Motor Co. Inc, "ASIMO by Honda. The World's Most Avanced Humanoid Robot," [Online]. Available: http://asimo.honda.com/. [Accessed September 2016].

[4] SoftBank Robotics, "Nao Evolution-Aldebaran," [Online]. Available: https://www.ald.softbankrobotics.com/en/press/press-releases/unveiling-of-nao-evolution-a-stronger-robot-and-a-more-comprehensive-operating. [Accessed September 2016].

[5] ASROB, "Mini-Humanoide. Asociación de Robótica UC3M," [Online]. Available: http://asrob.uc3m.es/index.php/Mini-Humanoide. [Accessed September 2016].

[6] RoboticsLab, "Welcome to Robotics Lab - Where technology happens," [Online]. Available: http://roboticslab.uc3m.es/roboticslab/. [Accessed September 2016].

[7] RoboSavvy, "Hitec - Robonova-1 Kit," [Online]. Available: https://robosavvy.com/store/hitec-robonova-1-kit.html. [Accessed September 2016].

[8] Robotis, "Bioloid - ROBOTIS," [Online]. Available: http://www.robotis.com/xe/bioloid_en. [Accessed September 2016].

[9] Github, "JavierIH/raider: Raider parts & programming," [Online]. Available: https://github.com/JavierIH/raider. [Accessed September 2016].

[10] J. I. Hernández, "Desarrollo de una plataforma robótica mini-humanoide con visión artifical," Madrid, 2014.

[11] CEA- Grupo Robótica, "CEABOT 2016," [Online]. Available: http://www.ceautomatica.es/sites/default/files/upload/10/CEABOT/index.htm. [Accessed September 2016].

[12] Cyberbotics, "Webots," [Online]. Available: https://www.cyberbotics.com/webots.php. [Accessed September 2016].

[13] C. Robotics, "Coppelia Robotics V-REP: Create. Compose. Simulate. Any Robot.," [Online]. Available: http://www.coppeliarobotics.com/index.html. [Accessed September 2016].

[14] OpenHRP Team, Humanoid Research Group, Intelligent Systems Research Institute, AIST, "About OpenHRP3," [Online]. Available: http://fkanehiro.github.io/openhrp3-doc/en/about.html. [Accessed September 2016].

[15] Universität Bremen, "SimRobot - Robotics Simulator," [Online]. Available: http://www.informatik.uni-bremen.de/simrobot/index_e.htm. [Accessed September 2016].

[16] C. Chambers, Towards reusable, extensible components, ACM Computing Surveys, 1996, p. 192.

[17] A. Brown, Building Systems from Pieces with Component-Based Software, vol. Constructing Superior Software, Indianapolis: MacMillan Technical, 1999.

[18] D. Carney and F. Long, «What do you mean by COTS? Finally, a usefull answer», IEEE Software, 2000, pp. 83-86.

[19] D. Bakken, J. Urban and P. Dasgupta, "Middleware," in *Encyclopedia of Distributed Computing*, Dordrecht, Kluwer Academics, 2001.

[20] OSRF, "About ROS," [Online]. Available: http://www.ros.org/about-ros/. [Accessed September 2016].

[21] OSRF, "ROS/Concepts - ROS Wiki," [Online]. Available: http://wiki.ros.org/ROS/Concepts. [Accessed September 2016].

[22] OSRF, "ros_control - ROS Wiki," [Online]. Available: http://wiki.ros.org/ros_control. [Accessed September 2016].

[23] OSRF, "Gazebo," [Online]. Available: http://gazebosim.org/. [Accessed September 2016].

[24] L. Nogueira, "Comparative Analysis Between Gazebo and V-REP," Campinas.

[25] Mathworks, "MATLAB - Mathworks," [Online]. Available: http://www.mathworks.com/products/matlab/. [Accessed September 2016].

[26] Mathworks, "Simulink - Simulation and Model-Based Design," [Online]. Available: http://www.mathworks.com/products/simulink/. [Accessed September 2016].

[27] Mathworks, "Robotics System Toolbox - Simulink and MATLAB," [Online]. Available: http://www.mathworks.com/help/robotics/. [Accessed September 2016].

[28] Blender Foundation, "About Blender," [Online]. Available: https://www.blender.org/about/. [Accessed September 2016].

[29] SourceForge, "MeshLab," [Online]. Available: http://meshlab.sourceforge.net/. [Accessed September 2016].

[30] OSRF, "ROS/Tutorials," [Online]. Available: http://wiki.ros.org/ROS/Tutorials. [Accessed September 2016].

[31] OSRF, "Gazebo : Tutorials," [Online]. Available: http://gazebosim.org/tutorials. [Accessed September 2016].

[32] OSRF, "Gazebo : Tutorial : Which combination of ROS/Gazebo versions to use," [Online]. Available: http://gazebosim.org/tutorials?tut=ros_wrapper_versions&cat=connect_ros. [Accessed September 2016].

[33] OSRF, "Gazebo : Tutorial : ROS communication," [Online]. Available: http://gazebosim.org/tutorials?tut=ros_comm&cat=connect_ros. [Accessed September 2016].

[34] Github, "rrbot - tutorial repository," [Online]. [Accessed September 2016].

[35] OSRF, "urdf/XML - ROS Wiki," [Online]. Available: http://wiki.ros.org/urdf/XML. [Accessed September 2016].

[36] Githtub, "bioloid_typea_description/typea.urdf," [Online]. Available: https://github.com/MathieuR/bioloid_typea_description/blob/master/typea.urdf. [Accessed September 2016].

[37] Github, "darwin_description/darwin.urdf," [Online]. Available: https://github.com/HumaRobotics/darwin_description/blob/master/urdf/darwin.urdf. [Accessed September 2016].

[38] OSRF, "Gazebo : Tutorial : Inertial parameters of triangle meshes," [Online]. Available: http://gazebosim.org/tutorials?tut=inertia&cat=. [Accessed September 2016].

[39] Github, "xavi-diaz/RAIDER," [Online]. Available: https://github.com/xavi-diaz/RAIDER. [Accessed September 2016].

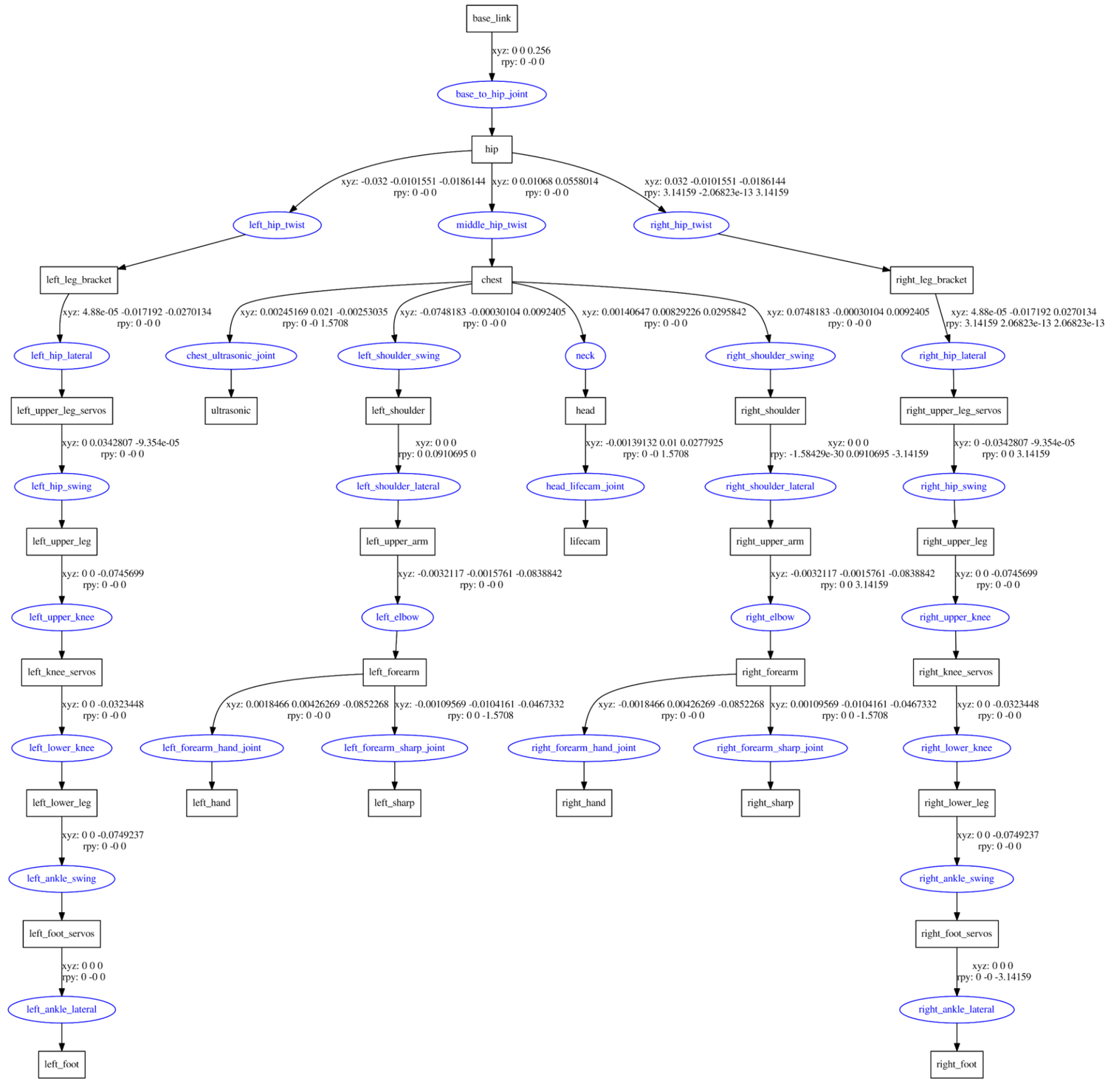
[40] Github, "raider/raider_motion.cpp at develop - JavierIH/raider," [Online]. Available: https://github.com/JavierIH/raider/blob/develop/code/opencm/motion/raider_motion.cpp. [Accessed September 2016].

[41] Youtube, "RAIDER walking in Gazebo!!," [Online]. Available: https://youtu.be/BhRjB-r8Wto. [Accessed September 2016].

[42] Agencia Tributaria - Gobierno de España, "Estimación Directa Simplificada - Agencia Tributaria," [Online]. Available: http://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresarios_individuales_y_profesionales/Rendimientos_de_actividades_economicas_en_el_IRPF/Regimenes_para_determinar_el_rendimiento_de_las_actividades_economicas/Est. [Accessed September 2016].

Appendix

In the next page a complete diagram with all the relationships between links (rectangles) and joints (ovals) of the file “Raider.urdf” can be consulted.


base_link
xyz: 0 0 0.256
rpy: 0 -0 0
base_to_hip_joint
hip
xyz: -0.032 -0.0101551 -0.0186144
rpy: 0 -0 0
xyz: 0 0.01068 0.0558014
rpy: 0 -0 0
xyz: 0.032 -0.0101551 -0.0186144
rpy: 3.14159 -2.06823e-13 3.14159
left_hip_twist
middle_hip_twist
right_hip_twist
left_leg_bracket
chest
right_leg_bracket
xyz: 4.88e-05 -0.017192 -0.0270134
rpy: 0 -0 0
xyz: 0.00245169 0.021 -0.00253035
rpy: 0 -0 1.5708
xyz: -0.0748183 -0.00030104 0.0092405
rpy: 0 -0 0
xyz: 0.00140647 0.00829226 0.0295842
rpy: 0 -0 0
xyz: 0.0748183 -0.00030104 0.0092405
rpy: 0 -0 0
xyz: 4.88e-05 -0.017192 0.0270134
rpy: 3.14159 2.06823e-13 2.06823e-13
left_hip_lateral
chest_ultrasonic_joint
left_shoulder_swing
neck
right_shoulder_swing
right_hip_lateral
left_upper_leg_servos
ultrasonic
left_shoulder
head
right_shoulder
right_upper_leg_servos
xyz: 0 0.0342807 -9.354e-05
rpy: 0 -0 0
xyz: 0 0 0
rpy: 0 0.0910695 0
xyz: -0.00139132 0.01 0.0277925
rpy: 0 -0 1.5708
xyz: 0 0 0
rpy: -1.58429e-30 0.0910695 -3.14159
xyz: 0 -0.0342807 -9.354e-05
rpy: 0 0 3.14159
left_hip_swing
left_shoulder_lateral
head_lifecam_joint
right_shoulder_lateral
right_hip_swing
left_upper_leg
left_upper_arm
lifecam
right_upper_arm
right_upper_leg
xyz: 0 0 -0.0745699
rpy: 0 -0 0
xyz: -0.0032117 -0.0015761 -0.0838842
rpy: 0 -0 0
xyz: -0.0032117 -0.0015761 -0.0838842
rpy: 0 0 3.14159
xyz: 0 0 -0.0745699
rpy: 0 -0 0
left_upper_knee
left_elbow
right_elbow
right_upper_knee
left_knee_servos
left_forearm
right_forearm
right_knee_servos
xyz: 0 0 -0.0323448
rpy: 0 -0 0
xyz: 0.0018466 0.00426269 -0.0852268
rpy: 0 -0 0
xyz: -0.00109569 -0.0104161 -0.0467332
rpy: 0 0 -1.5708
xyz: -0.0018466 0.00426269 -0.0852268
rpy: 0 -0 0
xyz: 0.00109569 -0.0104161 -0.0467332
rpy: 0 0 -1.5708
xyz: 0 0 -0.0323448
rpy: 0 -0 0
left_lower_knee
left_forearm_hand_joint
left_forearm_sharp_joint
right_forearm_hand_joint
right_forearm_sharp_joint
right_lower_knee
left_lower_leg
left_hand
left_sharp
right_hand
right_sharp
right_lower_leg
xyz: 0 0 -0.0749237
rpy: 0 -0 0
xyz: 0 0 -0.0749237
rpy: 0 -0 0
left_ankle_swing
right_ankle_swing
left_foot_servos
right_foot_servos
xyz: 0 0 0
rpy: 0 -0 0
xyz: 0 0 0
rpy: 0 -0 -3.14159
left_ankle_lateral
right_ankle_lateral
left_foot
right_foot