Université Libre de Bruxelles
Faculté des Sciences Appliquées
CODE - Computers and Decision Engineering
IRIDIA - Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Object Retrieval by a Swarm of Ground Based Robots Driven by Aerial Robots

The Design of a Multi-Engine and Multi-Robot Three-Dimensional Physics Simulator

Carlo PINCIROLI

Directeur de mémoire:

Prof. Marco DORIGO

Co-promoteur de mémoire:

Dr. Mauro BIRATTARI

Mémoire présentée en vue de l'obtention du Diplôme d'Etudes Approfondies en Sciences Appliquées

Année Académique 2006/2007

Abstract

This work deals with the design and the implementation of a novel multi-engine and multi-robot architecture of a three-dimensional physics simulator. Such architecture proves to be more flexible than existing state-of-the-art simulators and we have employed this tool to develop controllers for heterogenous swarms of robots. We have validated the functionalities of the simulator through an experiment of object retrieval which requires the cooperation of a flying robot to detect the target and of a ground-based robot to retrieve it and bring it to a goal area.

Acknowledgments

First of all, I would like to thank my supervisor Marco Dorigo for giving me the possibility to work at IRIDIA and to study in such an exciting field, Swarm Robotics. Also, I would like to thank all the partners of the Swarmanoid Project.

Furthermore, I would like to thank Mauro, Elio, Alex, Christos, Marco, Rehan, Vito, Alvaro and Xavier for all their suggestions and help. I would like to thank all the people of IRIDIA for making our lab everything but a working place.

Eventually, I would like to thank my family for continuously supporting me in my decisions, and for giving me a warm place I can always return to. Last but not least, I would like to thank Chiara for her love and care.

Contents

List of Figures

Chapter 1

Introduction

This work deals with the design and the implementation of a novel multi-engine and multi-robot architecture of a three-dimensional physics simulator.

The aim of this first chapter is to introduce the basic concepts of physics simulation and to underline its importance in robotics applications. Subsequently, the context in which the simulator has been developed, the Swarmanoid Project, is briefly presented. The final part of the chapter is dedicated to the statement of goals and original contributions of the thesis.

As stated by Ljung (1999),

> "When we interact with a system, we need some concept of how its variables relate to each other. With a broad definition, we shall call such an assumed relationship among observed signals a *model* of the system."

Mathematical models encode scientific knowledge about a particular system in a rigorous way. On the basis of the predictions that it provides, the correctness of a model can be verified, thus allowing to compare models and to keep the best ones (Gillies, 1993).

Simulators are computer programs that integrate modeling systems for which simple analytical solutions can not be found. Simulation allows one to quickly analyze a large number of model scenarios in situations in which their enumeration is practically impossible.

Furthermore, as reported by Frigg and Hartmann (2006),

> "In situations in which the underlying model is well confirmed and understood, computer experiments may even replace real experiments, which has economic advantages and minimizes risk (as, for example, in the case of the simulation of atomic explosions)."

During World War II, the first computers were built to make large-scale calculations feasible in a relatively short time: The Manhattan Project was the first to develop a model of the process of nuclear detonation in a computer.

Since then, the importance of simulators has grown rapidly, and nowadays proves to be an invaluable tool in many fields of science such as computational physics, chemistry, biology, economics, social sciences and engineering.

Obviously, simulators play an essential role in the development of robot controllers too. Simulated experiments are usually many orders of magnitude faster than real ones and simulated robots do not suffer hardware failures or battery exhaustion, unless this is explicitly desired. Moreover, simulations do not need to take into account purely technical issues such as calibration of sensors and actuators.

Despite these advantages, there is no guarantee that the controllers developed in simulation work as expected in real robots. Unfortunately, no general method to achieve a seamless transition exists.

A first tempting approach to tackle the problem is trying to model sensors and actuators of the robots as precisely as possible with the aim of rendering the difference between simulation and reality negligible. For instance, popular software engines, such as Open Dynamics Engine and Vortex, follow this idea. Anyway, it is practically unfeasible to perfectly simulate reality (Frigg and Hartmann, 2006), and the cost to make the model more faithful is often a substantial slowdown of the simulation.

At the opposite end of the spectrum, physics simulation could be avoided completely. Some sensors such as ground sensors, light sensors and infra-red proximity sensors can be implemented by sampling real readings taken from various positions and orientation with respect to other objects of the environment. The final result of this activity is a large numerical table containing the recorded samples that is fairly easy to import and fast to access at runtime. Nevertheless, obtaining such a table requires a long and error-prone work and with different robots the same sensors are likely to give significantly different readings. Furthermore, this technique does not appear applicable to sensors such as digital cameras.

A widespread complementary method to facilitate the difficulties to transfer controllers to real robots is adding noise to sensors reading and actuators outputs (Jacobi, 1997). It is a reasonable practice because real sensors and actuators are noisy. Additionally, noise injection softens the natural differences between sensors and actuators. The computational impact of noise addition is negligible but, thanks to it, resulting controllers are more robust and more easily transferable.

The Swarmanoid Project

Flexibility, robustness, decentralization and self-organization (Camazine et al., 2003) are the principles at the basis of *swarm robotics* (Bonabeau et al., 1999), a paradigm for developing robotics systems that in the last fifteen years is experiencing a growing interest among researchers. Swarm robotics aims at building swarms of small-scale and simple robots able to

collectively accomplish tasks such as exploration, object transportation, foraging and structure building taking inspiration from social insects. No robot in the swarm has a global knowledge of the environment or of the status of the swarm itself. Instead, each robot exploits only local information and a global behavior *emerges* from the interactions among the individuals.

The *Swarmanoid Project* is a futuristic project within the framework of swarm robotics that temporally and logically follows the *Swarmbots Project* — for a review of the project refer to (Dorigo et al., 2004) and (Mondada et al., 2004). As its predecessor, the Swarmanoid Project is a project funded by the European Commission. It involves five research laboratories across Europe: IRIDIA at Université Libre de Bruxelles in Belgium, Istituto di Scienze e Tecnologie della Cognizione at Consiglio Nazionale delle Ricerche in Italy, Laboratoire de Systèmes Robotiques and Laboratory of Intelligent Systems at Ecole Polytechnique Federale de Lausanne in Switzerland and Istituto Dalle Molle di Studi sull'Intelligenza Artificiale at Università della Svizzera italiana in Switzerland.

The main scientific objective of the Swarmanoid Project is the design, implementation, and control of an innovative distributed robotic system comprising heterogeneous, dynamically connected small autonomous robots so as to form a so called *Swarmanoid*. The Swarmanoid is comprised of a relatively large number of autonomous robots of three types:

Eyebots fly or attach to the ceiling. Thanks to their positioning capabilities and to the camera they are equipped with, these robots are able to quickly explore the environment and locate target objects and areas.

Handbots are intended to retrieve and manipulate objects located on walls, shelves, or tables. They can climb walls and obstacles by means of a rope they can shoot and attach to the ceiling.

Footbots are wheeled robots equipped with a rigid gripper used to assemble with other footbots, transport handbots or target objects.

The subject of the next sections is to present the hardware capabilities of the robots and to describe the goals of the Swarmanoid Project.

Hardware

A set of basic subsystems is common to all robots. In this way, while the development of the robots is facilitated, researchers are provided with a coherent tool set to build their controllers upon.

The common set of capabilities comprises the following:

- the main processor board: the Freescale i.MX31 ARM 11 processor, a low-energy 533 MHz

processor with a complete collection of subsystems such as USB host controller and integrated camera interface;

- the main core board built around the processor which provides 128 MB of DDR RAM and 32 MB of flash;
- a set of DsPIC 33 micro controllers for sensors and actuators;
- wireless communication in two fashions: WiFi, mainly intended for (relatively) long range inter-robot communication, and Bluetooth, for basic robot setup and short range inter-robot communication.

Footbot Hardware

In the Swarmbots Project the basic robot platform is the *s-bot*, a small scale robot with a full set of sensors and actuators. S-bots are able to connect to each other with a rigid gripper so as to form a *Swarmbot*. The success of the Swarmbots Project and especially of this platform suggested to base the footbot design on the s-bot. The evolution of the s-bot brought to the *marXBot*, a modular robot with higher computational power, more sensors with improved accuracy, better vision system, optimized battery usage and supporting WiFi, Bluetooth and RFID wireless communication.

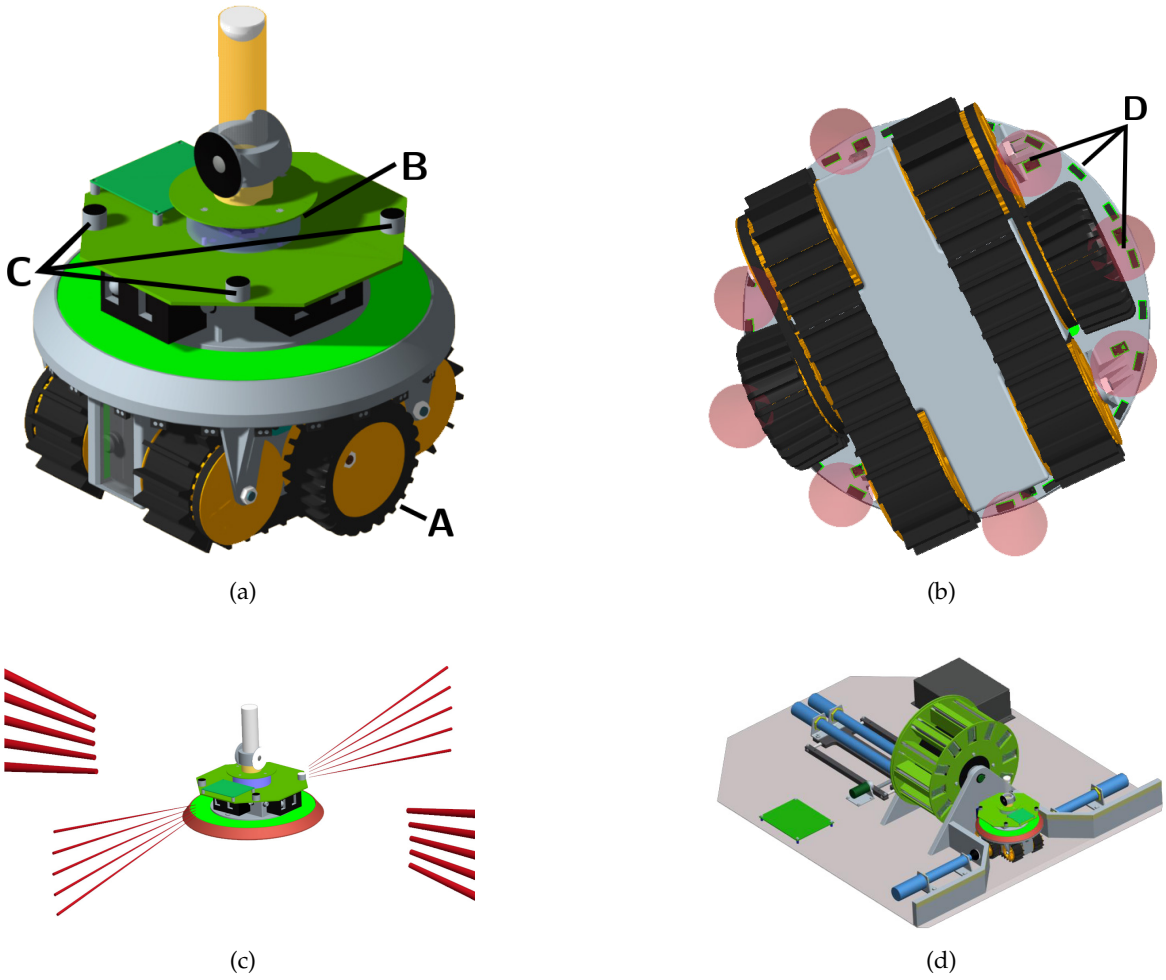
The footbot has been therefore based on the *marXBot* and Figure 1.1 details the most significant components of the footbot: the powerful treel drive; the 2.0 mega-pixels UXGA camera; the four microphones; the ring of infrared sensors for close presence and ground detection. The footbot is also equipped with a rotating long-range[1] infrared distance sensor. A big problem in collective robotics, particularly when performing long lasting experiments, is the frequency of recharge and the amount of time lost in it. To address this problem, the footbot has a hot-swappable battery. A super-capacitor keeps the robot alive while the battery is being exchanged. The exchange itself is performed autonomously by the robot thanks to the use of an automatic battery exchange station.

Handbot Hardware

The handbot is the most innovative robot of the Swarmanoid Project. It is intended to be a relatively low-cost, energy saving, light weight robot able to manipulate target objects located on the ground or on shelves.

At the time of writing, a full design of the handbot is not available due to the complexity of the issues to overcome. State-of-the-art climbing robots are by far less complex and face serious limitations with respect to the tasks the handbot is intended to perform. Anyway,

[1] Up to 1.5 m.



(a) (b)

(c) (d)

Figure 1.1: Footbot design: (a) the footbot and its components, (b) the ring of infrared sensors, (c) the rotating long-range infrared distance sensor and (d) the battery exchange station.

a tentative design has been studied and, as it can be recognized in Figure 1.2, the handbot should resemble a lobster.

Moving on the ground is made possible thanks to the intervention of footbots which, by gripping the handbot, transport it to the target area. The more footbots are needed to transport the handbot, the more difficult the task becomes.

Climbing is necessary to reach objects located on shelves. A rope with magnetic attach is the candidate method to move on walls. The rope is shot with a mechanical device that allows also to rewind it. The magnetic attach requires the ceiling to be ferro-magnetic, but this limitation in the environmental setup is not too constraining. When artificial adhesives such as those inspired from geckos (Menon et al., 2004) will be finally available, they will replace the magnetic attach. Anyway the shooting/attaching logic is expected not to change substantially.

Manipulation of objects requires a more evoluted gripper than the one mounted on the

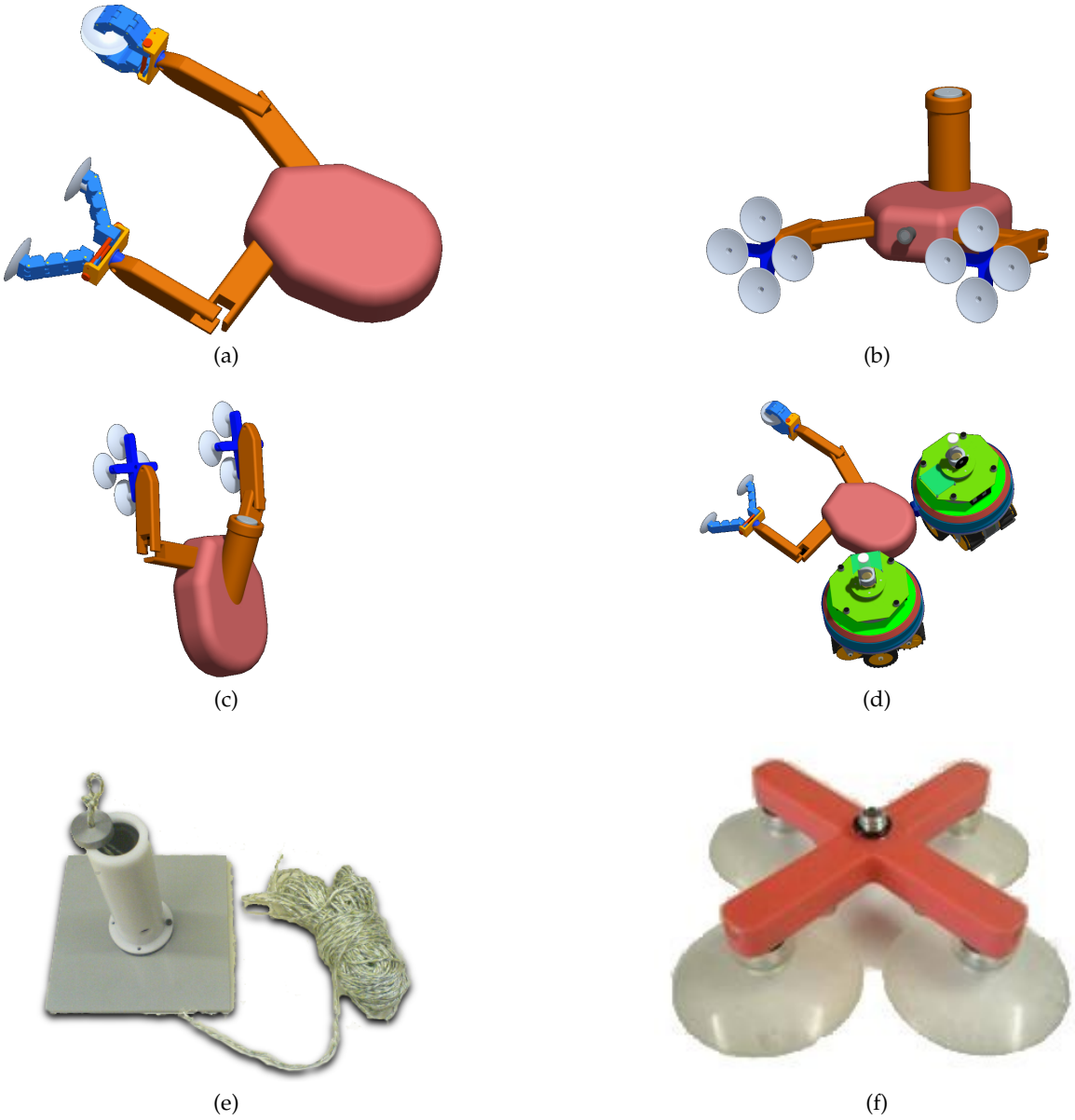


Figure 1.2: Handbot design: (a) the handbot with two vacuum suctions cups per hand, (b) with four, (c) an handbot stabilizing with hands while climbing a wall, (d) footbots attached to the handbot to transport it to a target area, (e) the rope launcher, and (f) a closeup of the vacuum suction cups.

footbot, and in fact we should better call it a rudimental hand instead of a mere gripper. The best solution proposed so far involves the use of vacuum suction cups because this allows the hands to be exploited both to grip and to stabilize while climbing a wall, if smooth enough. The use of two or four suction cups is currently under consideration.

The handbot will be equipped with a large number of sensors and actuators. A camera will allow vision to recognize target objects, while precise gripping along with obstacle

(a) (b)

Figure 1.3: Eyebot design: (a) the eyebot prototype hovering platform and (b) the designed aerodynamic ducts.

avoidance will be done thanks to several proximity sensors. Accelerometers and position and force sensors will be placed everywhere on the body of the handbot to make it feasible to sense its own current position and orientation. Eventually, a docking sensor will inform the robot of the gripping outcome.

Eyebot Hardware

Similarly to the handbot, the design of the eyebot platform is challenging. The current design status is quite advanced: Figure 1.3 shows a prototype of the robot that already presents the final hovering system.

Besides the usual requirements of low battery consumption, small size and light weight, the eyebot runs the serious risk of plunging and damaging. Therefore stability is a key feature of the flying system, and the available prototype already provides it at a good degree.

Furthermore, for added safety, the platform will be protected by a surrounding rigid aerodynamic duct to allow small collisions and also provide an increase in rotor efficiency. In addition, the duct provides a good medium to mount the external sensors and internal avionic equipment.

Battery power is expected to be saved with three alternative mechanisms: by attaching to the ceiling with a device similar to the magnet of the handbot, by landing temporarily in a clear space of no interest for the task being performed, or by changing batteries in a charging station analogous to the one of the footbot.

Eyebots will carry many sensors to help driving the robot, such as an inertial measurement unit and a differential pressure sensor. An high resolution omni-directional camera will be mounted on the robot to ease the exploration of the environment. A ring of colored LEDs will be available for basic visual communication among robots, and WiFi will be used

also to send visual information to all the robots in the swarm.

Project Goals

The ultimate goal of the Swarmanoid Projects is to show a new way of designing robotic systems that can live along with humans in human modified environments performing tasks of general utility.

Besides the development of the hardware platforms, the Swarmanoid Project aims at studying new control methodologies for the three types of robots following a kind of "holistic" approach. In the traditional methodology, first a controller for a single robot is developed, then swarm behavior with similar robots is inserted, and then finally interaction with the other types of robots is added. We will avoid as much as possible this approach, favoring the opposite one. Since one of main points of interest of the project is the heterogeinity among robots, controllers for the three families of robots are developed in parallel, so that the possible cooperative issues are tackled and solved in a smoother and more natural way. Furthermore, many interesting control issues are worth studying, such as the coordination of individual local perception by the robots into a coherent representation of the environment, and the study of mechanisms for adaptive task allocation in heterogeneous teams.

Another source of innovative challenges, which have never been addressed before, involves the study of communication in an heterogeneous swarm of robots. For instance, the emergence of communication in a robotic system in which hardware differences plays a central role is a completely new question. Moreover, implicit communicative strategies such as *stigmergy* (Grassé, 1959) have never been studied in heterogeneous swarms of robots. Finally, also studying explicitly how to efficiently share information among robots with so strongly different capabilities is a novel issue considered in the Swarmanoid Project.

Goal of This Work

In light of the importance of simulating reality in robotics applications discussed at the beginning of the chapter, and due to the fact that the hardware platforms are not available yet, the handiest tool to pursue the control goals of the Swarmanoid Project is a software simulator.

The aim of this work is to describe the architecture of the simulator that has been developed for the Swarmanoid Project, named *Swarmanoid Simulator*. The hardware characteristics of the robot platforms and the project goals trigger a number of challenges in designing a successful simulator.

First of all, multiple different robots must be simulated in a complex 3D environment. While footbots act mainly in a plain environment, the ground, the dynamics of handbots and eyebots can be faithfully modeled only in a three-dimensional space. Moreover, the

dynamics of the handbot is in principle very complex, due to the particular shape of the grippers and the way we intend to exploit them.

Furthermore, due to the heterogeneous expertise of the researchers working in the project, the simulator must offer to the user maximum flexibility both in the choice of the relevant aspects to simulate for a given experiment, and in the development of robot controllers following the preferred approach (for instance, behavior based (Arkin, 1998) vs. evolutionary techniques (Nolfi and Floreano, 2000)).

Another important design aspect is that performance is a critical issue, especially when controllers are obtained with evolutionary techniques. The simulator must be very fast and optimized, possibly letting the user decide where it is worth paying the highest simulation cost and, on the other hand, where approximation is allowed.

Fundamental importance has also the requirement of allowing the user to seamlessly transfer controllers developed in simulation into the real robots. This implies that on the one hand the same code must compile both for the simulator and for the robotic hardware platform without any corrective intervention by the researcher, and on the other hand that sensors and actuators of the robots need to be simulated properly, although, as already discussed, there exists no such thing as a perfect simulator.

In the following chapters, we present the resulting architecture of the Swarmanoid Simulator, which solves the issues here highlighted.

After illustrating the most popular and relevant simulators available for robotic applications, in Chapter 2 we also justify our choice of designing a completely custom architecture.

Subsequently, the discussion of Chapter 3 describes thoroughly all the relevant aspects of the architecture of the Swarmanoid Simulator. We claim that our architecture, being multi-robot, multi-physics engine and highly modular, proves to be more general than the existing ones, and that thanks to this generality it can be successfully reused for other projects.

Chapter 4 presents the simulator from the point of view of the user. We illustrate the software environment and show how to create a controller and configure the simulator to run an experiment. A demostrative experiment with a footbot and an eyebot is reported in which the cooperation of the two robots is required to grip an object and bring it to a target area. More specifically, the eyebot, after identifying the object in the arena, drives the footbot to it by means of a colored laser beam working as a pointing signal on the ground. The footbot, once reached the object, grips it and then it communicates to the eyebot the retrieval. Eventually, the eyebot leads the footbot to the target area with the laser beam.

Chapter 5 summarizes the main achievements of the presented work and outlines future directions for its improvement.

Chapter 2

State of the Art of Robot Simulation

Among the plethora of existing simulators, we have chosen to develop a completely new one. This chapter describes the existing simulator architectures and shows the motivations that pushed us to develop a new and more flexible architecture[1].

2.1 Existing Simulators

As explained in Chapter 1, computer simulation plays a central role in the development and evaluation of robotic systems. In fact, it can allow fast and extensive testing of robot hardware designs and control policies considering a large variety of environments.

On the downside, the intrinsic complexity of a (multi-)robot system and of the real-world environment it is supposed to act upon makes sometimes hard the design of realistic simulation models to derive sound evaluations and predictions of the robotic system under study.

2.1.1 Discrete-time and continuous-time simulation

Computer simulation can be classified in several ways. In the domain of multi-robot systems the large majority of the simulators adopts a dynamic model in which the system changes in response to input signals, and makes use of random number generators to model chance or random events.

In *continuous-time simulation* the model of the simulated system evolves according to a continuous time flow and the simulation is stepped in small time increments (usually constant) that discretize the time flow.

On the other hand, in *discrete-event simulation*, time evolution is triggered by the happening of events inside the system.

[1]The contents of this chapter are inspired by the Swarmanoid Project deliverable D3 on the simulation platform, which we have contributed to write.

In a continuous-time simulation system, evolution is based on the numerical solution of differential equations. At each iteration step, the state equations are solved and the results are used to change the state of the system and the output of the simulation. This simulation model is particularly appropriate to simulate systems, or parts of systems, that are explicitly governed by the laws of physics. This is usually the case of multi-robot systems embedded in complex real-world environments, undergoing robot-to-robot and robot-to-environment physical interactions.

2.1.2 Simulators for multi-robot systems

The comparative study by Seugling and Rölin (2006), who took into account a number of features and performance indices of eight popular physics engines free for non-commercial use, shows significant differences among the considered engines. According to the study, *Novodex*, which is a commercial software produced by AGEIA, is the best physics engine, closely followed by *Open Dynamics Engine (ODE)*, which is an open source software distributed with the GNU Lesser General Public License. On the other hand, it is well-known that the commercial *Vortex* software from CM-labs provides unsurpassed precision, stability, and accessories.

A brief review of some among the most interesting tools available for physics-based multi-robot simulation in 3D environments here follows. All the considered tools have implementations for both Linux and Windows machines. A more general and complementary discussion on development tools for multi-robot systems can be found in the recent overview paper of Kramer and Scheutz (2007), where several simulators and programming interfaces for robotics have been compared and evaluated with respect to available features, usability, and impact. A list of available resources for development and simulation in robotics can be found at `http://robotica.cz/software/en`.

Player/Stage/Gazebo (Gerkey et al., 2003) (`http://playerstage.sourceforge.net`)

> The Player Project aims at producing free software to enable research in robot and sensor systems. Software code is developed by an international team of robotics researchers and used at many laboratories around the world (the web site gets an average of 2000 downloads per month). *Player* is a multi-threaded robot device that provides full access and control of a robotic platform and of all its sensors and actuators. At the moment Player supports a wide variety of existing mobile robots and accessories. Client programs can be written in any language, run both locally or remotely, and connect to the server via TCP sockets. Player supports multiple concurrent client connections to devices, offering the possibility for distributed and collaborative sensing and control. *Stage* is a multiple robot simulator. It simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment. Various sensor

models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry. Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware. *Gazebo* is a multi-robot simulator that extends Stage's capabilities for 3D outdoor environments. It generates both realistic sensor feedback and physically plausible interactions between objects using ODE's libraries for the simulation of rigid-body physics. Gazebo presents a standard Player interface in addition to its own native interface. Controllers written for the Stage simulator can generally be used with Gazebo without modification (and vice-versa). The simulated scenario is input using an XML syntax. User visualization is based on the use of OpenGL libraries.

USARSim (Carpin et al., 2007) (`http://usarsim.sourceforge.net`)
Urban Search and Rescue Simulation (USARSim) is a high fidelity multi-robot simulator that was originally developed in the context of the search and rescue (SAR) research activities of the Robocup contest. It is now becoming one of the most complete general-purpose tools for robotics in research and educations. Its development is driven by a large community of researchers. It builds upon a widely used and affordable state-of-the-art commercial game engine, the Unreal Engine 2.0, produced by Epic Games, which provides high accuracy of physics simulation, a number of geometrical and physical models, good computational speed, and usage flexibility. Robots are fully customizable and can be effectively controlled by a client program through a TCP socket connection. USARSim provides a large collection of robot models, including wheeled, legged, flying, and underwater ones, and of fully configurable sensors and actuators with associated noise models. Quantitative evaluations show a close correspondence between results obtained within USARSim and with the corresponding real world system or sensor. USARSim can be interfaced with the Player middleware to seamlessy control real robots and to the Mobility Open Architecture Simulation and Tools framework (MOAST) which provides a hierarchical control system. USARSim comes with a number of high definition test scenarios taken from indoor and outdoors situations ranging from the NIST test arenas for urban SAR to speedways.

Webots (Michel, 2004) (`http://www.cyberbotics.com`)
Webots is a commercial robotic simulator developed by the Cyberbotics Ltd. It has an ODE-based accurate physics simulation and provides several models for real robots such as Sony Aibo, Khepera, or Pioneer2. The robots and the environment are described using the VRML standard for graphical models, extended by nodes for the Webots elements, sensors, and physical attributes. Mobile robots with any physical characteristics can be designed, including flying, wheeled, and legged ones. Controllers can be programmed in C++ or Java and connected to third party software through a TCP/IP interface. An extensive library of tunable sensors and actuators is provided,

including distance and global positioning sensors, compass, cameras, radio transmitters, incremental encoders, etc. A powerful graphical visualization is realized with the use of OpenGL libraries. Webots numbers a large community of users across the globe and undergoes continual updating.

UberSim (Go et al., 2004) (`http://www.cs.cmu.edu/~robosoccer/ubersim`)
This simulator is developed at Carnegie Mellon with the intent to create an open source high-fidelity simulator for dynamic robot soccer scenarios that enables rapid development of control systems that can be transferred to real robots with a minimum of overhead. UberSim makes use of ODE to provide realistic dynamics including motions and physical interactions. UberSim is targeted towards providing parametrized robot classes that are easy to extend and reconfigure. Exploiting ODE's capabilities, it provides the definition and use of robot shapes and actuators which are generic enough to allow simulation of a wide range of robot types. Own robots can be modeled by programming their structure in C classes. UberSim has a client/server based architecture, where clients communicate with the server over TCP sockets. In the current release only few sensors are defined and no graphical interface is provided for user interaction at simulation time.

Breve (Klein, 2002) (`http://www.spiderland.org`)
It is a simulation package designed for realistic simulations of large distributed and artificial life systems in continuous 3D worlds with continuous time. Simulations are written by defining the behaviors and interactions of agents using a simple object-oriented programming language called *Steve*. Breve makes use of the ODE physics engine libraries to provide facilities for rigid body simulation, collision detection/response, and articulated body simulation. It is intended to permit the rapid construction of complex multi-agent simulations in realistic physical environments. Breve includes an OpenGL display engine that allows observers to manipulate the perspective in the 3D world and view the agents from any location and angle. Users can interact at runtime with the simulation using a web interface. Multiple simulations can interact and exchange individuals over the network. Code development is mainly done by one single researcher. Breve is the simulation package used in the context of the *SwarmRobot* project aimed at building large-scale robotic swarms (`http://www.swarmrobot.org`).

2.2 Why Yet Another Simulator?

The Swarmanoid Project as described in Chapter 1 presents many challenges for the development of a simulator. We opted for a fully custom design, instead of using as reference platform one of the available simulators described in the previous section, such as Gazebo, USARSim, or Webots. In the following we describe the motivations supporting our choice.

First of all, in order to simulate the specific characteristics of the robots composing the Swarmanoid and their sensors and actuators, even when using an existing simulator platform we need in any case to (re)implement from scratch the majority of the modules. For instance, none of the mentioned simulators include modules that could help to simulate the handbot climbing along the vertical dimension by shooting a rope that gets magnetically attached to the ceiling. Therefore, in the case of choosing building on an existing simulator, we would have found ourselves in the position of implementing from scratch and/or heavily adapting most of the simulation modules, somehow vanishing in this way the benefits of using a preexisting simulator, but at the same time being also forced to adapt to a general software structure selected by a third party.

Computational performance is always a critical issue for a simulator. It is reasonable to believe that a simulator based on a good ad hoc design can provide better performance than a general purpose one. In our case running times are expected to play a critical role for the usability of the simulator. In fact, from one hand, we will have to deal with multiple highly heterogeneous robots interacting in realistic abstractions of real-world scenarios, and from the other hand, we plan to heavily use computationally-demanding evolutionary algorithms to synthesize robot controllers. Therefore we do need to have at hand a simulator showing a fully satisfactory trade-off between high simulation fidelity and fast execution time. The most computationally intensive part of a realistic 3D simulator consists in the calculations carried out by the physics engine. The more accurate is the model adopted to deal with physical interactions, the more computational resources are required. In most of the available simulators only one physics engine is made available. Accuracy is selected on a global level. At the local level it is only possible to temporarily switch on and off, or change the level of accuracy of the operations of the physics engine for a specific object (e.g., for a temporarily inactive object, or for an object of minor interest).

On the other hand, in our design we devised an innovative solution which can provide a better accuracy-computation trade-off. We allow the use of multiple physics engines in the same simulation run, with each engine taking care of a specific portion of the space and/or of a specific subset of the robots. For instance, if in a certain simulation experiment the role of the eyebots is marginal, such that it does not make much difference whether their behavior is simulated with high physical accuracy or not, physics of eyebots at the ceiling level can be managed by a simplified engine while a more accurate physics engine can be used for the footbots, optimizing in this way the use of computational resources without losing relevant information.

Moreover, the architecture provides a common developing platform for all the researchers cooperating in the Swarmanoid Project. Thanks to its flexibility, each user can define its own models for robot actuators and sensors, or even realize a completely new physics engine. The plugin system implemented in the simulator allows a straightforward integration of the

newly coded modules in the framework.

Chapter 3

The Swarmanoid Simulator

Chapter 2 pointed out the reasons why a new and more flexible architecture is required for the development of robot controllers in the Swarmanoid Project.

The topic of this chapter is the actual architecture of the simulator. After a first overall view of the way the architecture is divided into modules and of the mutual connections among them, the discussion continues detailing the internals of each module.

3.1 Architecture

From a very abstract point of view, the traditional simulation algorithm involves the following operations:

1. Initialize Simulation
Create the simulated arena, place all simulated objects in their initial positions, create robot controllers, initialize physics engine and visualization.

2. Arena Visualization
The status of the arena is shown on the screen and/or written to a file.

3. Run Controllers
Each robot controller is executed. Internally, a controller reads the sensor inputs and, according to the control logic, outputs values to the actuators.

4. Physics Update
The physics engine updates position and heading of each simulated object in the arena and resolves collisions.

5. Next Simulation Step
Return to point 2.

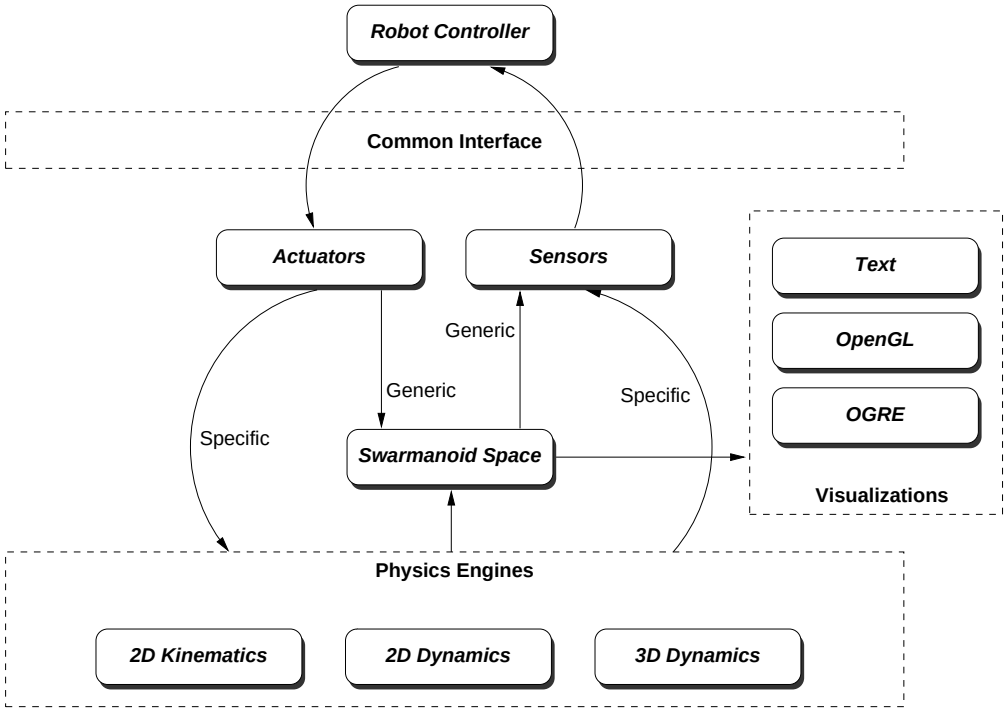


Figure 3.1: Overall architecture of the Swarmanoid Simulator.

Usually the assumption is made that all simulated objects live inside a space, which is accessed both by the physics engine and the visualizations. Moreover, it can be stated that the nature of the physics equations coded in the physics engine defines such space: a 2D physics engine, that is a physics engine that implements bidimensional equations, constrains the space to be bidimensional. Visualisation can be either 2D or 3D, but the actual actions of the robots are nevertheless bidimensional. Likewise, a 3D physics engine, such as one based on Vortex or ODE, will push towards a 3D space.

To allow the user to optimize the experiment by focusing on the accurate simulation of the relevant aspects while letting the others be approximated (see Chapter 1), the Swarmanoid Simulator has been designed to support the possibility of running different physics engines independently during an experiment. For example, the user can select footbots to run in a 2D physics engine, while having handbots and eyebots updated by another one, or even other two, separated 3D physics engines. This key feature has been obtained by decoupling the space in which all objects live from the physics engines. Such global space goes under the name of *Swarmanoid Space*. Figure 3.1 depicts the architecture of the Swarmanoid Simulator. The 3D Swarmanoid Space can be found at the center of the picture. Physics engines access it to update its status and visualizations display it.

Robots interact with the environment through simulated sensors and actuators. As already discussed in Chapter 1, the ease to transfer controllers developed in simulation to real

robots heavily depends on how sensors and actuators have been modeled. The architecture of the Swarmanoid Simulator does not favor any approach (physics-based or sample-based), thus letting the developer choose the best approach in each case.

Anyway, a classification of sensors and actuators has been studied to provide a coherent structure for their development. Some sensors and actuators rely on physics equations: for instance, the torque sensor computes the torque between the body and the turret of a footbot, while wheels update the position of a footbot according to Newton's Laws. Other sensors, such as the camera, simply rely on positional information to compute their readings. Likewise, some actuators, such as the LEDs actuators, do not need any physical information to perform their actions.

Sensors and actuators that rely on physics models must be reimplemented for each different physics engine. For this reason, we call them *specific*. On the other hand, sensors and actuators that do not need any interaction with the physics engines are termed *generic*.

A strong attention has been paid to design a highly modular architecture. Since the Swarmanoid Project is a project that involves many researchers, modularity is a key feature to ensure cooperation and reuse of code. Each box in Figure 3.1 has been implemented as a plugin. The user can code his own version of the module, and easily inform the system about its existence. Compatibility and interoperability are guaranteed by the interfaces that will be described in the following sections.

On the basis of this general presentation of the architecture, it is now possible to illustrate the abstract simulation algorithm of the Swarmanoid Simulator:

1. Initialize Simulation

1-1 Create the simulated arena

1-2 Place all the simulated objects in their initial positions in the Swarmanoid Space

1-3 Create robot controllers

1-4 Initialize physics engines and assign robots to them

1-5 Initialize visualizations

2. Arena Visualization (*for each visualisation*)

2-1 Display the status of the Swarmanoid Space.

3. Physics Update (*for each physics engine*)

3-1 Each robot controller is executed. Internally, a controller reads the sensor inputs and, according to the control logic, outputs values to the actuators.

3-2 The physics engine updates position and heading of the assigned simulated object and resolves local collisions. Then, it translates local coordinates to global ones.

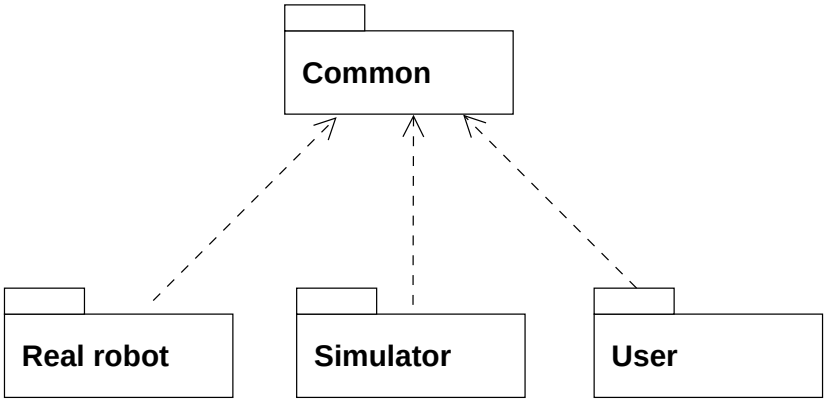


Figure 3.2: The package diagram of the Swarmanoid Simulator.

4. Next Simulation Step

4-1 Return to point 2.

3.2 Code Organization

Figure 3.2 reports the software packages in which the code has been divided and their mutual dependencies. The division in packages has been designed with the intentions of maximizing code reuse while keeping the functionalities logically separated.

Package *Common* contains classes and functions used in the other packages, therefore they all depend on it. In *Common* we can find string utilities, common definitions, logging facilities, mathematical definitions and functions, and so on. The most important part of this package is the definition of the common control interface, which is covered in Section 3.3.

Package *Real robot* contains the compilation environment and the software interfaces of the three hardware platforms. Its purpose is to provide a tool set to compile software for the real robots. In this package, the common control interface is implemented using functions in the robot APIs. Since the hardware platforms are not yet available at the time of writing (for more details refer to Chapter 1), in this document we will not describe the internals of this software package.

The actual code of the simulator, the achitecture of which is the topic of this document, is contained in package *Simulator*. In this package the common control interface is implemented to provide simulated sensors and actuators. Moreover the Swarmanoid Space, the physics engines and the visualizations as depicted in Figure 3.1 are all included in this software package. The core class of the simulator is `CSimulator`, as it is possible to observe in Figure 3.3. Its role is to implement the abstract algorithm illustrated in Section 3.1. The other interfaces of the architecture are thoroughly described in Sections 3.4 through 3.6.

Finally, a package has been created to contain all user code. The aim of package *User*

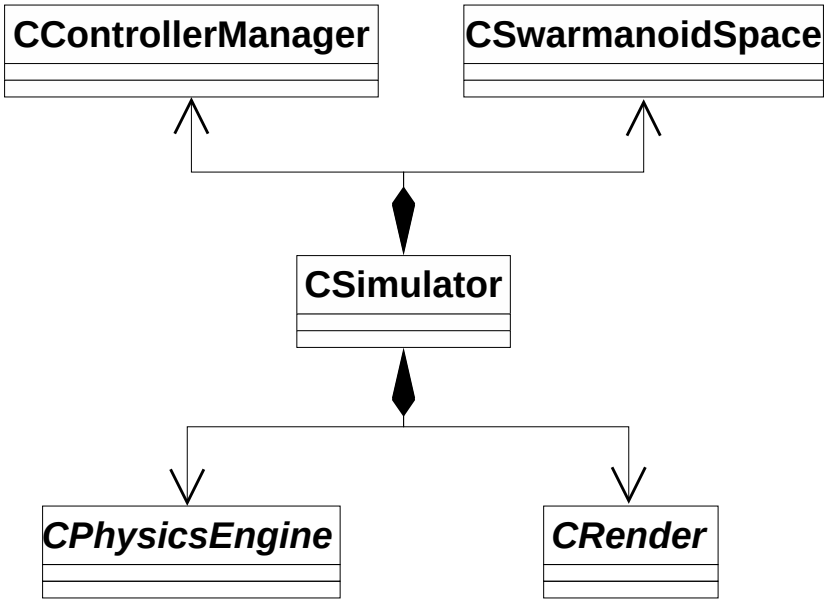


Figure 3.3: The class diagram of the core classes of the Swarmanoid Simulator.

is to collect in a structured way controllers and other software developed by the users so that knowledge, experiments and solutions are shared in a natural way. On the other hand, to prevent a user to erroneously damage the work of other users by deleting and/or modifying files in an arbitrary and uncontrollable way, each user is provided a subpackage to develop his own pieces of software. In this way package *User* allows cooperation among the developers while keeping them independent. The internals of package *User* are presented in Section 4.1.

3.3 Common Interface

One of the requirements of the Swarmanoid Simulator is a seamless transition between simulation and reality. In other words, it should be possible to develop a controller in simulation and directly transfer it to the real robots, without any change in the structure of the code. Anyway, this does not imply that the behavior produced by a given controller in simulation will correspond to what observed with the real robot. The successful transfer of developed behaviors must be ensured by a careful modeling of the robot features. Here, we just refer to the possibility to directly compile a controller against both the simulation and the real environment. For this purpose, an interface (referred to as *Common Interface*) completely independent from the simulation engine has been developed.

Figure 3.4 presents a schematic class diagram of the Common Interface, which encompasses a robot interface, a controller interface and sensors and actuators interface. The Common Interface provides virtual access to all the features necessary to develop a controller in a

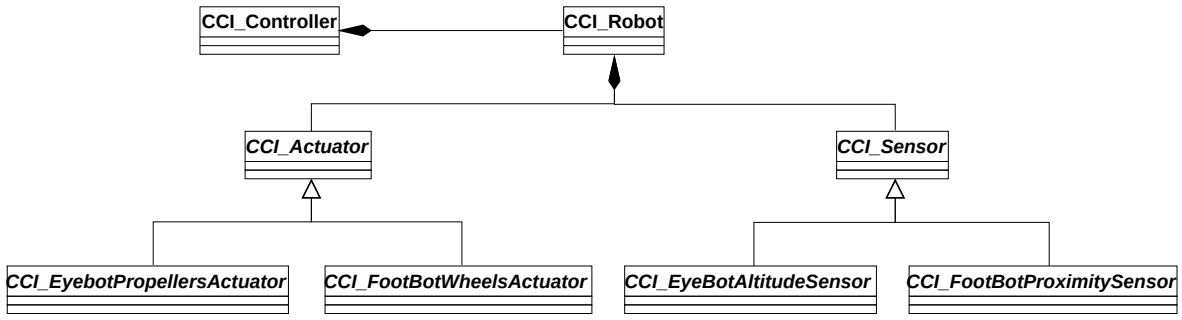


Figure 3.4: A schematic class diagram of the Common Interface

transparent way with respect to the simulated or the real world. In the following, we briefly describe each component of the Common Interface.

Robot Common Interface

The robot common interface is undifferentiated with respect to the different robotic platforms developed within the Swarmanoid Project (see Figure 3.4). This interface provides an abstraction for the modalities to access the different features of a robot. From the control point of view, a robot is a set of devices (usually referred to as sensors and actuators) that should be used to define the rules that govern the behavior of the robot. A common interface for the robot is useful to define how a controller can access to its devices. Moreover, it defines also how to initialize the robot itself along with all the devices useful for a certain experiment. In fact, as pointed out in Chapter 1, the robots developed within the Swarmanoid Project are rather complex artifacts, composed of different mechatronics parts that need special initialization procedures whenever they are going to be used. Otherwise, if certain devices are not necessary for a given experiment, they can be left uninitialized, allowing also to reduce the energy consumption and to increase battery lifetime. It is therefore important to provide an interface to the robot that allows to initialize and access only those devices that are actually used in a certain experiment.

Controller Common Interface

The controller interface is an abstraction that provides common methods for the interaction between control rules and the simulation environment or the real robotic platforms (see Figure 3.4). The controller does not have direct access to the devices of the robot, but it should request them to the robot interface. This choice is suggested by the need to provide a generic framework for the different control design methodologies (e.g., behavior based or evolutionary robotics design).

The controller interface basically provides three main methods: `Init`, `ControlStep` and `Destroy`. Every user-defined controller must implement these functions, which are

executed in the same way on the simulated and on the real robots.

Sensors and actuators common interfaces

Sensors and actuators constitute respectively the inputs and the outputs of a controller. Within the common interface, these interfaces provide the abstraction of any devices that can be implemented on the real robots (see Figure 3.4). The generic sensor and actuator interface does not refer to a single sensor (for instance, one of the two wheels of the footbot), but it rather refers to a set of sensors/actuators (for example all proximity sensors around the turret of a footbot). In this way, it is possible to define group-level functions, such as `GetAllSensorReadings` or `SetAllActuatorOutputs`, which in many cases allow to speed up the computation and simplify modeling the devices in simulation.

From the general interfaces, we derive the detailed interface for each particular device. In Figure 3.4 only some example interfaces are displayed, such as the one for the footbot wheels or the one for the eyebot altitude sensor. At this level, each interface corresponds to a particular device. We will also develop interfaces that refer to particular modalities of accessing a device. For instance, the omni-directional camera can grab images that can be used in many different ways. More specifically, we have developed an interface for color blob extraction that abstracts a given image filtering algorithm, which is implemented in hardware and is simulated in some way. Similarly, it could be possible to implement an interface for displaying a preferential direction with the footbot's LED ring, which corresponds in hardware to a particular activation of the LEDs, while in simulation it could just store and modify a value for the desired direction.

To be able to upload the controllers developed in simulation onto the real robots, it is necessary that all sensors and actuator interfaces an implementation for the real robot. Typically, a single implementation is sufficient, but for complex devices such as the camera, there can be different algorithms implementing the same feature which could be tested in parallel. On the contrary, in simulation usually there are different ways of simulating a particular device, depending on the desired accuracy. For this reason, various implementation of the same interface can be developed. Moreover, the simulation of certain devices depends on the physics engine currently used. As a consequence, the implementation of a specific device is necessarily polymorphic, because the actions required to simulate it change with the physics engine.

3.4 Swarmanoid Space

The Swarmanoid Space is the 3D space where the simulation is performed. All the simulated robots as well as all the features that characterize the experimental arena (e.g. holes, walls, obstacles, light or sound sources) are stored in it, so the Swarmanoid Space acts as a common

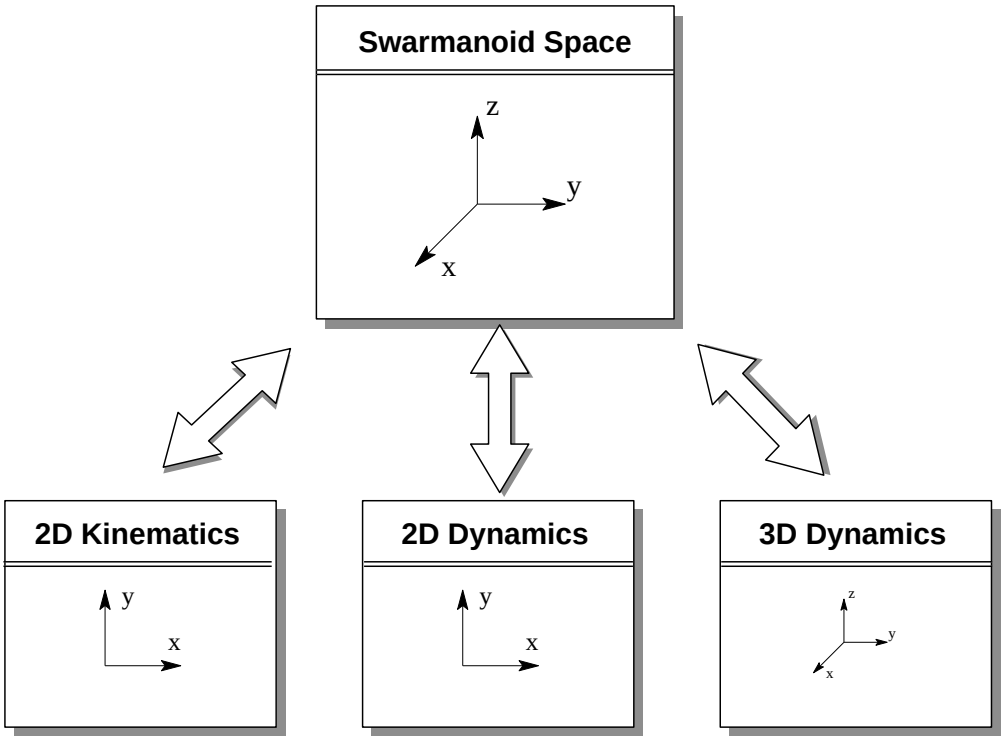


Figure 3.5: The Swarmanoid Space provides a global 3D coordinate space for all the entities. The physics engines perform their calculations in a local reference frame, and then local coordinates are transformed into global ones.

3D reference frame. Section 3.5 explains that physics engines possess a local coordinate frame to update the entities assigned to them. All the calculations are performed with respect to the local coordinate frame. Subsequently, local coordinates are translated to global ones (e.g. the Swarmanoid Space, see also Figure 3.5).

Therefore, thanks to the Swarmanoid Space, every simulated object (referred to as *entity*) has a unique global position in the space. In addition, the Swarmanoid Space stores the global simulation clock. At each tick of the clock, every physics engine is called once, and then the visualisations display the new status.

The translation mechanism, along with time synchronization with the global clock, ensures coherence. Visualisations exploit the information in the Swarmanoid Space to provide an overall picture of the running experiment.

Furthermore, generic sensors and actuators are allowed to access information in the Swarmanoid Space. The proximity sensors, for example, are provided functions to retrieve all the entities with a certain distance from a point in the space or an entity. Other functions have been implemented to check physical occlusions from a point to another. These last functions, anyway, call the corresponding functions in the physics engine(s) which an entity is bound to.

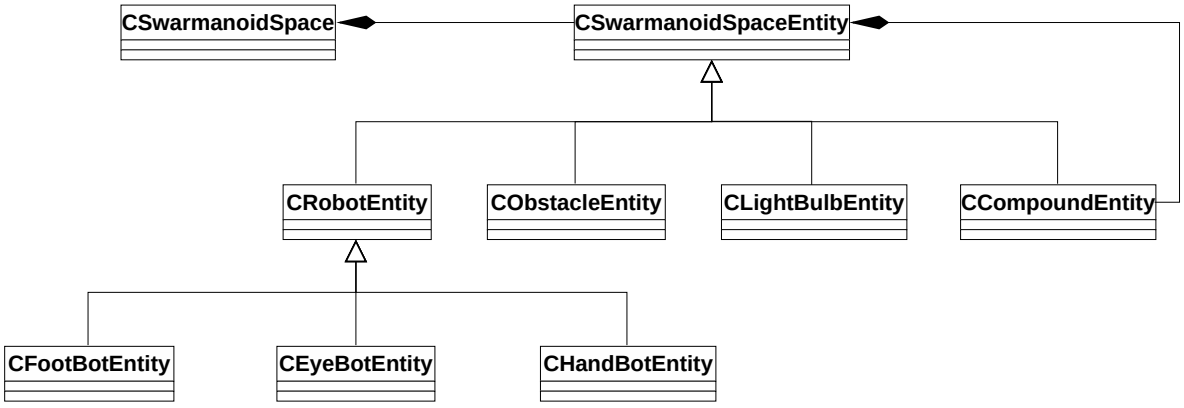


Figure 3.6: A schematic class diagram of the Swarmanoid Space.

In order to ease and optimize the access to such information, entities are indexed into *scene graphs*. Scene graphs are tree-like data structures used to store two types of information in an optimized way: *positional* information and *structural* information.

Positional information is encoded in an *oct-tree* (Salomon, 2004), a well-known tree structure which partitions the coordinate space so that entity location and range searches can be performed with a complexity of $O(\log n)$, rather than $O(n)$[1], which is the complexity of an approach based on a simple list of entities.

When robots are connected to each other, e.g. a footbot grips an handbot or a target object, we talk about structural information. Trees are the best solution in this case too. The root of the tree is defined as the gripped object, and the gripping object is a leaf. A complex tree structure emerges from the fact that preys and handbots can be gripped by more than one footbot at the same time and footbots can grip also other footbots. In a footbot-only structure, the footbot which is not gripping anything is the root of the tree. Furthermore, we assume that handbots manipulate only one object at a time. This way to implement gripping excludes loops but for the kind of experiments we think to study this is an acceptable limitation.

Entities are organized in a class structure schematically depicted in Figure 3.6. The base class is `CSwarmanoidSpaceEntity`, which contains the most abstract definition of an entity. It basically reduces to identifier, position and heading of the entity and an association to the physics engines in charge of updating it. In general, these classes only store data that is useful for visualization or modeling of generic sensors and actuators. All the physics characteristics such as mass, speed or acceleration are stored inside class `CPhysicsEngineEntity` (refer to Section 3.5).

`CRobotEntity` adds to the base class the association to an object of type `CCI_Controller`, which is explained in Section 3.3. `CFootBotEntity`, `CHandBotEntity` and `CEyeBotEntity`

[1]Where n is the number of nodes in the tree.

are an extension of `CRobotEntity`, which store the specific data of each robot, such as the LEDs color and the gripper aperture status for the footbot.

As already stated, the Swarmanoid Space stores also other arena objects as entities. For instance, `CLightBulbEntity` is a model of the light emitter. Sound emitters, obstacles, and similar objects can be modeled analogously.

The structure tree of assembled entities is implemented in class `CCompoundEntity`. This class represents a compound object, that is a set of connected entities. When entities are assembled into a compound, such object acts as a whole. Connections are assumed to be rigid. Although in principle any object can be connected to each other, from an implementation point of view, only objects possessing a gripper (such as footbots or handbots) or objects possessing a gripping strip (such as footbots and handbots) are allowed to connect.

3.5 Physics Engines

In general terms, the component in a simulation program that computes how physical objects move and interact with each other according to the laws of classical physics is termed a *physics engine*.

Implementing a realistic physics engine is not a trivial task due to the intrinsic instabilities and limitations in the equations used to describe the dynamics of complex bodies in real-world environments[2].

When using a physics engine, an object is explicitly modeled in terms of attributes such as mass, velocity, friction forces, and elasticity. Its shape can be abstracted by using regular primitives from solid geometry or can be represented by triangle meshes, which can in principle be used to represent any shape. Clearly, the more accurate and faithful is the description, the more complex and computationally expensive is the solution of the motion equations.

The behaviour of a physics engine consists of two main phases, *collision detection* and *dynamic simulation*. At each time step, all the possible collisions and constraints are considered by the *engine solver* and new positions, velocities and accelerations are calculated accordingly after integration of the equations of motion.

There exists a number of different ways to represent and implement motion equations, friction forces, collision detection, and equation integration, resulting in a number of different physics engines.

The main novel aspect of the architecture of the Swarmanoid Simulator is the possibility to implement physics engines as plugins and run them in parallel. The degree of flexibility and generality deriving from such design choice allows the user to decide which physics

[2] An overview of physics engines together with a collection of references can be found in (Seugling and Rölin, 2006).

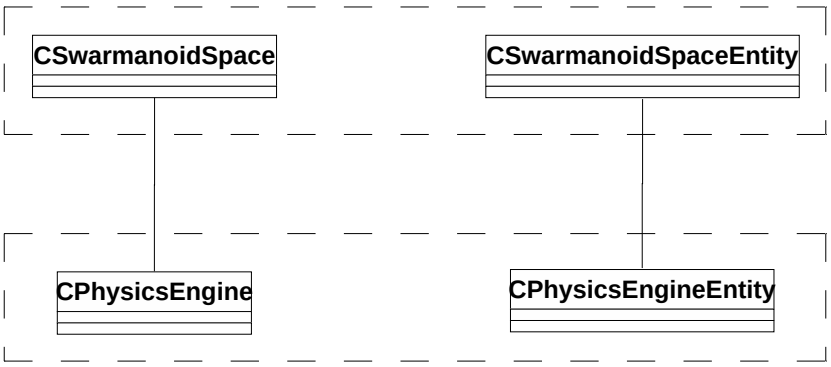


Figure 3.7: The class diagram of the physics engine interface related to the associated classes of the Swarmanoid Space.

engine to use to simulate a set of robots, so that experiments can be optimized and accuracy is ensured where it is really needed.

Section 3.4 explains that the global reference frame where all simulated objects act is the 3D Swarmanoid Space. A physics engine has the task of updating the set of robots assigned to it. In the architecture no hypothesis is done about the kind of rules that the physics engine should follow. Three-dimensional as well as bidimensional (even monodimensional) engines can be freely inserted in the system. Each engine let the robots act in a local (i.e. specific to the engine) coordinate frame whose dimensionality is decided by the physics equations implemented in it. Subsequently, a geometrical transformation maps local coordinates into global ones, as schematically depicted in Figure 3.5.

The fact that multiple physics engines can run independently from each other raises the issue whether to handle or not collisions among robots belonging to different physics engines. The simplest and cleanest solution is to consider every physics engine as a closed world, thus preventing robots associated to different engines to collide because they live in two "parallel worlds". If, by chance, two robots belonging to different physics engines happen to occupy very close positions in the 3D space, they simply interpenetrate. Even if this seems to be a big constraint, or, worse, a design mistake, in fact this feature provides a way to further optimize experiments. Knowing in advance that two groups of robots will not interact (e.g. collide or grip each other) in an experiment, as it is usually the case of footbots (acting only on the ground) and eyebots (mainly flying or attached to the ceiling), there is no reason to assign both groups to the same physics engine. Moreover, less robots assigned to an engine means faster collision detection, thus leading to overall better performance. On the other hand, if we want two groups of robots to interact, such as footbot and handbots, the two groups must belong to the same physics engine.

As Figure 3.7 shows, the software interface of the physics engines is essential. In substance it consists of the two thin interfaces `CPhysicsEngine` and `CPhysicsEngineEntity`.

The first, `CPhysicsEngine`, defines the main functions of an engine, that basically are

concentrated in the `Update` method. At each time step, the simulator calls this method to ask the engine to calculate new positions and headings for the entities assigned to it. The association with `CSwarmanoidSpace` allows the physics engine to retrieve useful information from the scene graphs, if needed.

On the other hand, the role of `CPhysicsEngineEntity` is to provide a basic definition of the entity as seen inside a physics engine. This interface has been designed to be absolutely minimal to keep the constraints it imposes on the development of a physics engine null or negligible. For this reason, no default link with `CPhysicsEngine` has been defined in any of the two interfaces, leaving complete freedom to the developer to implement the best linking strategy (e.g. lists, maps, or trees). Each physics engine entity is associated to its own alter ego in the Swarmanoid Space through an association with the corresponding `CSwarmanoidSpaceEntity`.

3.5.1 2D Kinematic physics engine

At the lower-end in terms of realism, there are physics engines that are kinematic. In these physics engines only first-order (velocity-driven) dynamics is considered, objects are abstracted by their center of mass, collisions are purely elastic, and friction forces are not taken into account.

We have developed a bidimensional kinematic engine because of its high performance. Despite its simplicity, most of the dynamics of footbots are acceptably simulated, thus making it almost always useless to use a more accurate engine. This is obviously false for eyebots and handbots, whose dynamics strongly depend on gravity and inertial effects. Nevertheless, when the simulation of their dynamics is not important for the purpose of the experiment, a kinematic model of handbots and eyebots proves to be very useful to shorten the running time.

The kinematic engine let the robots move on a plane that can be placed in the 3D space by specifying the main axis it should be perpendicular to (either x, y or z) and its distance from the origin. Figure 3.8 depicts a situation in which three instances of the kinematics engine have been created: one for the ground, one for the wall, and one for the ceiling. All the instances are run independently from each other by the architecture.

Figure 3.9 depicts the simple geometrical models for collision detection. Footbots are composed by a big central circle (the body), two smaller circles (the wheels) and a square (the gripper). The model of an eyebot is even more essential: just a circle. Eventually, handbots do not have a defined stable shape yet, although it is highly probable that they will resemble a lobster (see Figure 1.2 and refer to Chapter 1 for more details about the design status of the handbot robotic platform). For this reason, only the most basic model, a rectangle, has been so far implemented.

Gripping in this engine is supported in a very basic way. Footbots are provided a gripper

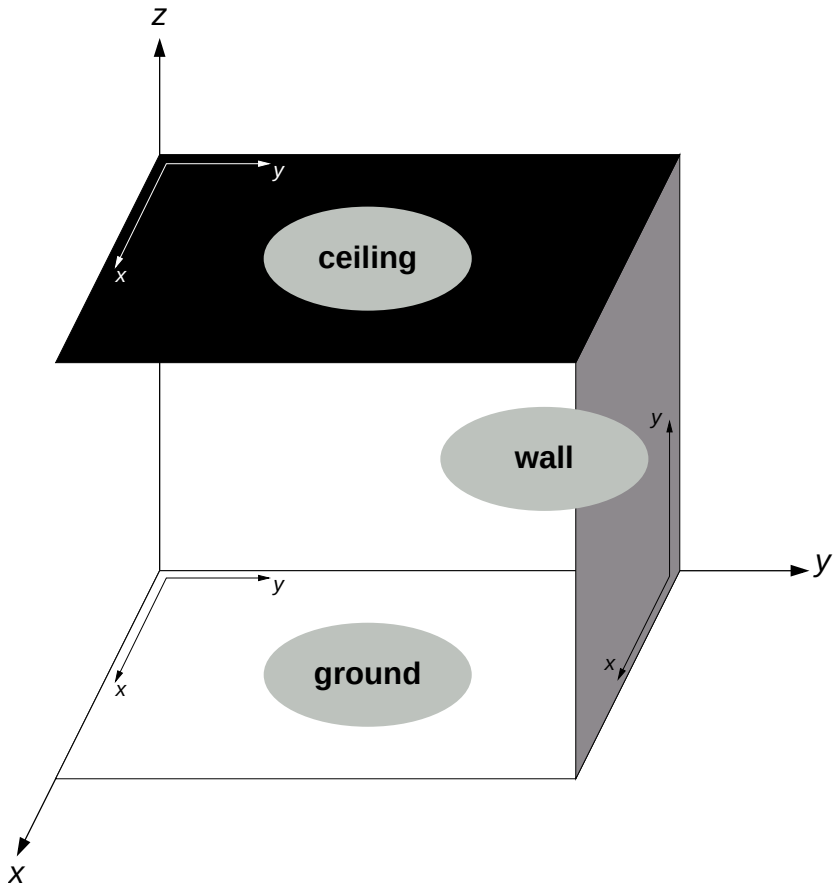


Figure 3.8: How planes can be placed in the Swarmanoid Space when using the 2D kinematics engine.

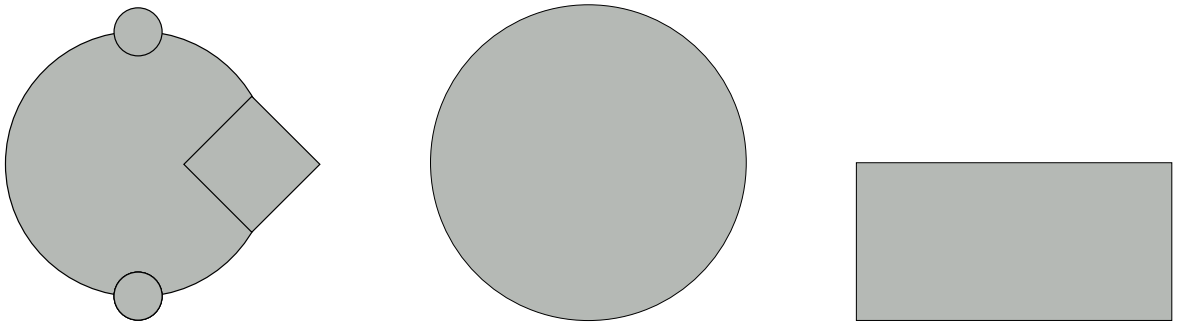


Figure 3.9: 2D kinematic collision models: (a) footbot, (b) eyebot, and (c) handbot.

that can only attach to target objects termed *preys* which possess a special gripping strip around their bodies. Furthermore, footbots cannot grip other footbots. This is a simplistic approach to maintain performance good. Also, rigid body dynamics require masses and accelerations to be simulated properly, but, as we have already illustrated, in a kinematic

physics engine acceleration and masses are not taken into account.

3.5.2 2D Dynamic physics engine

It is currently under development a bidimensional dynamic physics engine that will have most of the features of the kinematic engine, such as collisions models and placement logic, with the addition of a complete implementation of Newton's Laws. Rigid body dynamics will be also fully supported to allow arbitrary gripping structures thus fully taking advantage of the structural scene graph constructed in the Swarmanoid Space (see Section 3.4). Many ideas about rigid body dynamics and specific sensors/actuators will be inspired by TwoDee (Christensen, 2005), a bidimensional dynamic simulator developed for the Swarmbots Project.

3.5.3 3D Dynamic physics engine

A fully featured three-dimensional dynamic physics engine is also being designed. It will be based on ODE, an open source software, and it will model in an accurate way footbots on rough terrain, handbots with their complex climbing and manipulating abilities, and eyebot hovering dynamics.

For eyebots, an ODE based physics engine supporting the prototype robot is already available in the OMISS simulator[3], though it has not been integrated in the architecture yet.

3.6 Visualizations

The evaluation of an experiment requires a mechanism to visualize what is happening in the arena, often measuring the value of some interesting parameters.

The most widespread way to show the actions of the robots is simply to graphically render the arena and its contents either as the experiment is carried out or in the form of a video clip to watch after the end of the experiment. On the other hand, non-graphical visualizations (i.e. providing structured numerical data) play an important role to allow a quantitative evaluation of an experiment. Furthermore, the organization and the nature of the recorded information depends on the focus of the experiment.

The role of proper visualization is therefore as important as the one of physics simulation itself. Moreover, the degree of flexibility that it demands is definitely higher, because non-graphical visualization logic dramatically depends on the focus of the experiment, which in turns dictates the interesting values to record.

The modular architecture of the Swarmanoid Simulator has been designed to let the developer insert visualizations as plugins in the same way sensors, actuators and physics en-

[3] Available online at `http://iridia.ulb.ac.be/wiki/index.php/OMiss_-_The_ODE_based_simulator`.

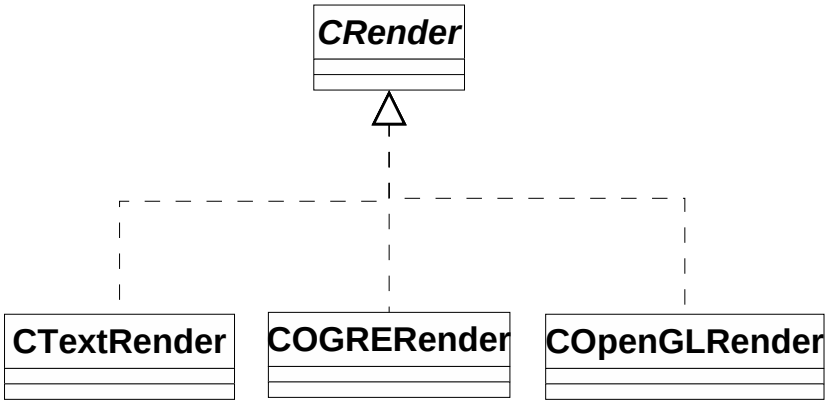


Figure 3.10: Visualization code class diagram.

gines are treated. Similarly, a software interface named `CRender` has been kept as thin as possible to leave the developer free to define the most appropriate visualization logic.

`CRender` basically consists of two methods: `Draw` and `Terminate`. The former is in charge of actually displaying the status of the Swarmanoid Space. The method is called in the simulation loop, and everything happening inside is completely transparent to the rest of the simulator. On the other hand, the latter method, `Terminate`, is in charge of intercepting the desire of the user to stop the simulation before the maximum clock tick count has been reached, for example when the main graphical window is closed.

Figure 3.10 displays the class diagram of the visualization code. In the following sections we will present these modules.

3.6.1 Text-based visualization

The simplest visualization offered by the Swarmanoid Simulator is a plain text tabular representation of the objects in the space. `CTextRender` offers the most basic set of functions on top of which a more refined output can be obtained by extending the class at will.

The class writes the table to the screen or to a file. A sample output of the file is reported in Figure 3.11. As shown, the output is organized in a simple tabular format, perfectly suitable to be input to mathematical programs such as Matlab, Octave, Excel or Gnuplot, or also string filters such as `grep` and `awk`.

The columns of the table, as it is easy to understand by reading Figure 3.11, have following meaning:

 # clock The current clock tick count. The actual duration of the clock tick in milliseconds is specified in the XML file as described in Section 4.2.

Entity type The type of the entity, such as a footbot, a handbot, an eyebot, a block (i.e. obstacle), and so on.

```
# clock Entity type Entity id   X    Y    Z    Alpha Beta Gamma
10      Block       wall1       0    5    1.5  0     0    0
10      Block       wall2       5    0    0.25 0     0    0
10      Block       wall3       10   5    0.25 0     0    0
10      Block       wall4       5    10   0.25 0     0    0
10      Block       ceiling     1    5    3    0     0    0
10      Block       column1     2    0    1.5  0     0    0
10      Block       column2     2    10   1.5  0     0    0
10      Block       obstacle    4.5  1.5  0.25 0     0    45
10      Block       table_plane 8    4    0.5  0     0    0
10      Block       table_leg1  8.97 4.47 0.25 0     0    0
10      Block       table_leg2  8.97 3.52 0.25 0     0    0
10      Block       table_leg3  7.03 4.47 0.25 0     0    0
10      Block       table_leg4  7.03 3.52 0.25 0     0    0
10      Block       le1_base    4    7    0.25 0     0    0
10      Footbot     fb_l1       3    6    0    0     0    45.5
10      Footbot     fb_l2       3    7    0    0     0    0.455
10      Footbot     fb_l3       3    8    0    0     0    -44.5
10      Footbot     fb_l4       5    6    0    0     0    135
10      Footbot     fb_l5       5    7    0    0     0    -180
10      Footbot     fb_l6       5    8    0    0     0    -135
10      Eyebot      yb1         2.97 3    1.5  0     0    -178
10      Eyebot      yb2         4.97 5    2.25 0     0    -178
10      Eyebot      yb3         4.97 3    1.5  0     0    -178
10      Eyebot      yb4         5    7.03 2.25 0     0    91.8
10      Handbot     hb1         1    3    0.25 0     0    0
10      Handbot     hb2         1    5    0.25 0     0    0
10      Handbot     hb3         1    8    0.25 0     0    0
```

Figure 3.11: Sample output of the basic text visualization.

Entity id The identificative name of the entity.

X, Y, Z The position of the entity in the Swarmanoid Space.

Alpha, Beta, Gamma The value of the Euler angles storing the orientation of the entity expressed in degrees.

3.6.2 Graphical visualization: OpenGL

Graphical rendering plays an important role to visually check the outcome of an experiment, for instance to verify that the behavior of a robot corresponds to what expected. Furthermore, the ability to produce video clips proves precious when presenting an experiment to an audience.

The OpenGL renderer provides a simple three-dimensional visualization of the simulated arena. Code is based on OpenGL, as the name suggests, a fast library for 3D rendering used in many different fields, from modeling to video games. The internal code structure is purposely straightforward and the robot models are built composing basic primitives such as boxes, cylinders and spheres.

This renderer is suitable to quickly display an experiment when there is no need of fancy graphical presentation. "Visual" debugging of some critical parts of the system is possible simply by extending this renderer to display more information. For example, the collision detection library implemented for the 2D kinematic physics engine was debugged allowing the OpenGL renderer to display also the collision models of the robots.

Some sample frames taken with this visualization are displayed in Figure 3.12. Although code structure is simple, this module provides a good number of useful functions. For instance, it is possible to move the camera by dragging the mouse and to record the current camera position. The user can store up to ten camera viewpoints and later recall them by pressing on the keyboard CTRL-0 ... CTRL-9. Furthermore, video frames can be stored on the hard disk, thus allowing to mount video clips with an external program. To speed the computation up on slow computers, window size can be set, and textures and shadows can be toggled.

3.6.3 Graphical visualization: OGRE

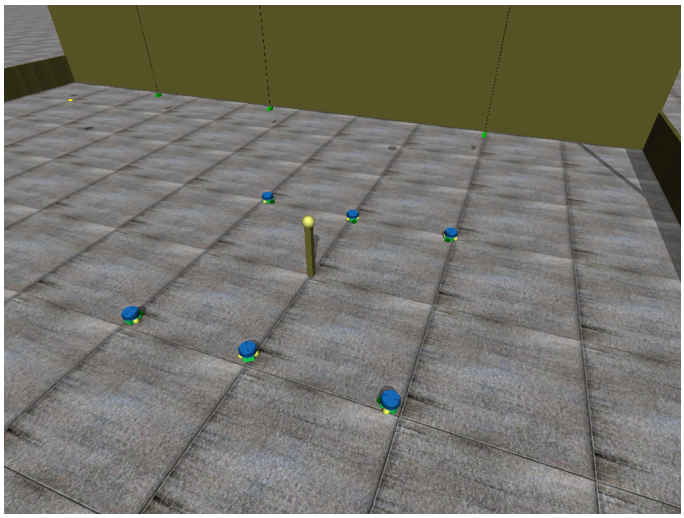
The basic graphics offered by OpenGL are not always enough for the development of robot controllers. In fact handbots, with their highly complex dynamics involving wall climbing and object manipulation, often require a proper visual modeling to satisfyingly distinguish the movements of the hands. Devising a precise model of the handbot in OpenGL is a complicated task, due to the limited set of primitives it provides. Furthermore, it is often useful to be able to select robots, open windows displaying their status, moving them in other locations, and similar actions.

The need of more accurate graphics and the desire to add more refined functions to the graphical user interface, while keeping the OpenGL renderer simple, led to the design of a completely new renderer.

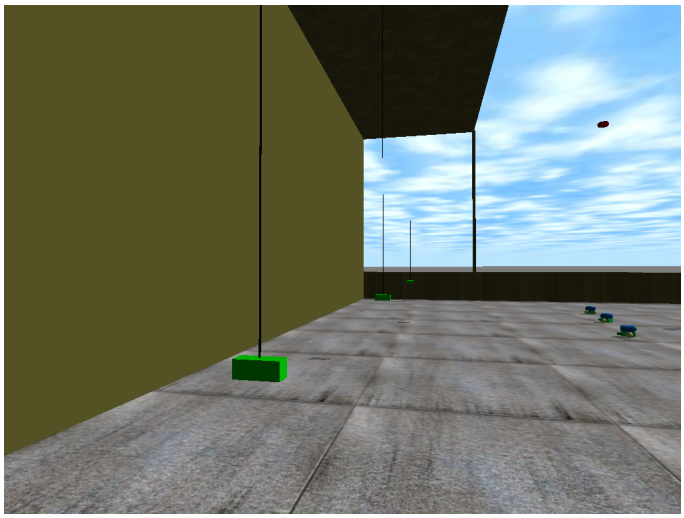
The OGRE library is a 3D graphics library built on top of OpenGL. It provides higher level functions than those found in OpenGL. For instance, it is possible to import a 3D mesh model of an object, access in an evoluted way the input from keyboard and mouse, and build seamlessly intuitive graphical interfaces.

For these reasons, the new renderer has been implemented using the OGRE library. Some sample frames are displayed in Figure 3.13. They show the 3D models of footbots (Figure 3.13a), handbots (Figure 3.13b) and eyebots (Figure 3.13c), along with the high level graphical user interface.

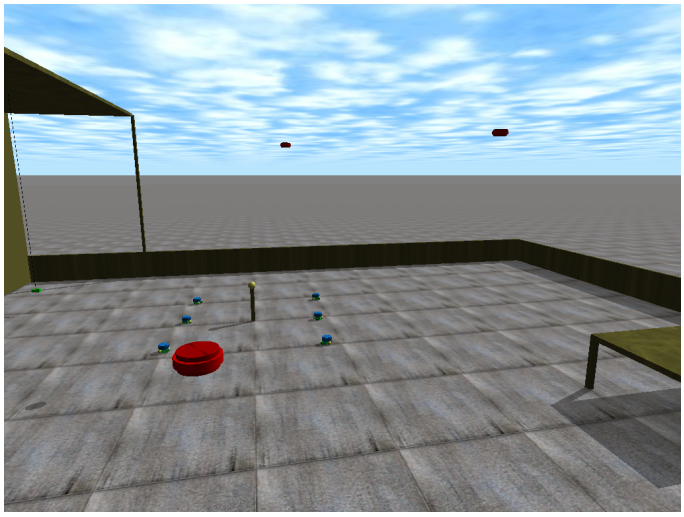
The OGRE renderer, similarly to the OpenGL one, allows to store frames to mount video clips. In addition, it offers the possibility to select a robot and watch its status (sensor readings, speed, etc.). Moving a robot is not possible yet but it is a feature under development at the time of writing.



(a)

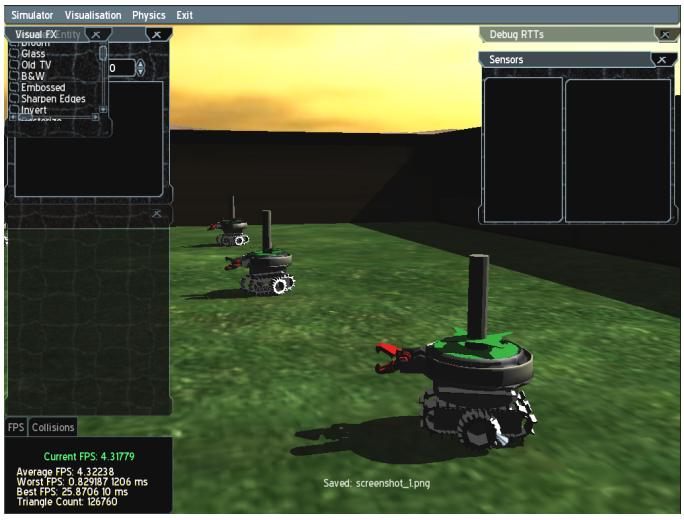


(b)

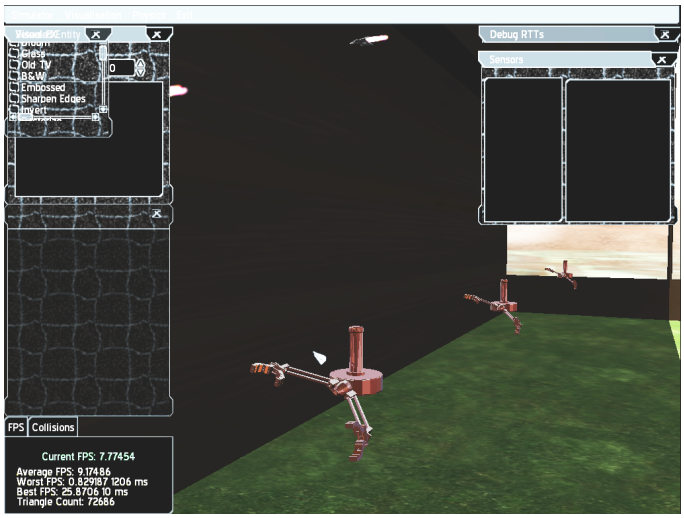


(c)

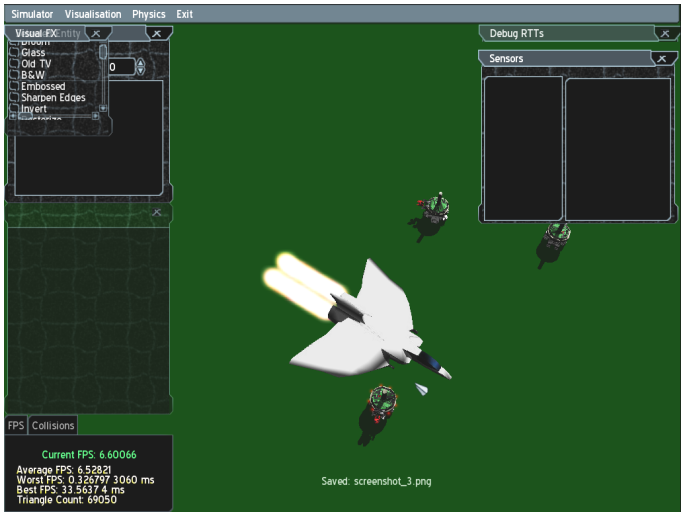
Figure 3.12: OpenGL visualization example: (a) footbot, (b) handbot, and (c) eyebot.



(a)



(b)



(c)

Figure 3.13: OGRE visualization example: (a) footbot, (b) handbot, and (c) eyebot.

Chapter 4

The Simulator at Work

Showing the simulator at work is the aim of this chapter. The subject of Section 4.1 is the user environment, that is how the user can create a controller for a simulated robot. Section 4.2 describes the structure of the XML file that configures the simulator to run an experiment. Finally, Section 4.3 reports the results of an experiment of prey retrieval by a footbot and an eyebot in which cooperation between the two robots is required to accomplish the task.

4.1 User Environment

When the Swarmanoid Simulator is downloaded from the repository for the first time, the initial step to start using it is to build the code.

An automatic script called `build_framework` is provided in the base directory of the framework. By running it, all the code in packages *Common*, *Real robot* and *Simulator* is compiled. When the compilations finishes, the user is asked by the script to create a personal directory in package *User* to store his own code. Subsequently, the newly created folder is also stored in the common software repository of the Swarmanoid Project. In this way, future contributions of the new user will be automatically shared with the other users.

A fresh creation of the user directory already contains many sample files to ease the first attempts to use the framework. For example, the user can execute the provided set of sample experiments to verify the success of the installation. Moreover, the sample controllers and the configuration files allow the user to start modifying an already working environment, allowing him to learn by doing.

Furthermore, the compilation environment is based on Autotools, a portable tool set for automating the compilation on different computer platforms. As a consequence, compiling the framework and subsequently adding controllers and other code can be done very easily. Moreover, the libraries used to code the simulator and the choice of Autotools make it feasible to port the framework on many platforms. At the time of writing only Linux is supported, but we plan to port our code to other popular operating systems such as MacOS X

and Microsoft Windows.

4.1.1 The Contents of the User Directory

The user directory contains the following files and directories:

configure.ac
: This file configures an automatically created script called `configure` that checks for the presence of the required libraries and functions, and initializes the basic Makefile structure.

Makefile.am
: A file named `Makefile.am` is present in each directory that stores code files. It contains the list of files to compile so that Automake can create the actual Makefiles used to build the code.

bootstrap.sh
: This is a custom script provided to wrap the call to the standard script `configure`.

build.sh
: This script internally calls `bootstrap.sh` and then runs the compilation of the code.

simulation_main.cpp
: This file contains the `main` function of the simulator. We have decided to insert this file in the directory of the user so as to allow him to modify it at will, for example wrapping it with other analysis tools.

simulation_build
: This directory contains all the user code compiled for the simulator.

real_robot_build
: This directory contains all the user code compiled for the real robot.

controllers
: This directory contains all the controllers of the user.

It is important to note that the described directory structure can be modified at will by the user, although for some particular modifications a big deal of technical experience is a requirement to accomplish this task flawlessly. Anyway, if the user respects this directory structure, the automatic tools provided by the framework let him focus on the development of the controllers.

4.1.2 Writing a New Controller

The creation of a new controller is the topic of this section. The configuration of the simulator to run an experiment is covered in Section 4.2.

Thanks to the code examples provided by default at the moment of creating the user directory, writing a new controller is not a complicated task. Usually it is enough to choose an appropriate sample controller as a template and modify it. Here, anyway, we assume to write a new controller from the very beginning.

The following listing shows a sample header file for the a new footbot controller that uses only the wheels:

```
#ifndef _CSAMPLEFOOTBOTCONTROLLER_H_
#define _CSAMPLEFOOTBOTCONTROLLER_H_

#include "ci_controller.h"
#include "footbot/ci_footbot_wheels_actuator.h"
#include "config.h"

class CSampleFootBotController: public CCI_Controller {

      // Associations
   private:
      CCI_FootBotWheelsActuator* m_pcFootBotWheelsActuator;

      // Attributes

      // Operations
   public:
      virtual int Init ( const TConfigurationTree t_tree );
      virtual void ControlStep (  );
      virtual void Destroy (  );
};

#endif
```

The interface is very simple: we have two methods, called `Init` and `Destroy`, that respectively initialize and destroy the object. They do not coincide with the constructor and the destructor, because the construction/destruction logic is left to the framework to manage;

anyway, from the point of view of the user, `Init` contains the code to retrieve references to sensors and actuators, to create structures such as neural networks, and so on. Furthermore, this method receives as input an XML data structure that is the configuration subtree specified for this controller in the configuration file (refer to Section 4.2 for details).

On the other hand, `ControlStep` is in charge of reading sensor input and, on the basis of it, choosing the next actions, that is writing values to the actuators.

The following listing shows a skeleton of implementation of the header that also demostrates how to obtain a reference to actuators and sensors:

```
#include "sample_footbot_controller.h"

using namespace swarmanoid;
using namespace ahss;
using namespace std;

int CSampleFootBotController::Init(const TConfigurationTree t_tree)
{
   m_pcFootBotWheelsActuator =
      (CCI_FootBotWheelsActuator*)(GetRobot(  )->
         GetActuator( "footbot_wheels" ));
   return CCI_Controller::RETURN_OK;
}

void CSampleFootBotController::ControlStep (  )
{
   m_pcFootBotWheelsActuator->
      SetFootBotWheelsAngularVelocity( 0.314, 0.628 );
}

void CSampleFootBotController::Destroy (  )
{}

REGISTER_CONTROLLER( CSampleFootBotController,
                     "sample_footbot_controller" )
```

As explained in Section 3.3, an object of type `CCI_Robot` is a container of sensors and actuators. Therefore in method `Init` a reference to the wheels actuator is obtained by means of method `CCI_Robot::GetActuator`. The string value passed to it is a type label associated

with the actuator at the moment of registering it; as it will be shown in Section 4.2, such type label is also used in the XML configuration file to let the simulator initialize the actuator.

In order to inform the framework of the existence of the controller, we have to register it. The registration logic is the same for all the plugins in the architecture, such as sensors, actuators, physics engines and visualizations. The last line of the listing performs the registration, passing to the macro the name of the created class and an identificative label that, as it will be shown in Section 4.2, will be used as XML tag to reference this controller.

4.2 Definition of an Experiment

To configure the simulator for running an experiment, it is sufficient to write the wanted parameters in an XML file, the structure of which is explained in this section.

The following listing shows that the XML file is divided into several parts, each of which is in charge of configuring a subsystem of the simulator:

```
<ahss-config>

  <!-- ********************************* -->
  <framework>
    ...
  </framework>

  <!-- ********************************* -->
  <controllers>
    ...
  </controllers>

  <!-- ********************************* -->
  <arena size="10,10,3" optimization="2D" >
    ...
  </arena>

  <!-- ********************************* -->
  <engines>
    ...
  </engines>

  <!-- ********************************* -->
```

```
  <arena_physics>
    ...
  </arena_physics>

  <!-- ********************************* -->
  <visualisations>
    ...
  </visualisations>

</ahss-config>
```

The order in which the tags appear in the file is not important, as the simulator analyzes them by name.

Tag `<framework>` defines a section in charge of initializing the parameters that interest the architecture in general. For example, it may appear as shown:

```
<framework>

  <clocktick>100</clocktick>
  <maxclock>500</maxclock>
  <controller_path>
    /path/to/swarmanoid_dev/user/pincy/simulation_build/\
    controllers/mycontroller1/.libs:/path/to/swarmanoid_dev/\
    user/pincy/simulation_build/controllers/mycontroller2/.libs
  </controller_path>

</framework>
```

Tags `<clocktick>` and `<maxclock>` together define the total duration of an experiment. In fact, `<clocktick>` sets the duration of a clock tick in milliseconds[1], while `<maxclock>` indicates the maximum number of clock ticks[2] after which the experiment is considered finished. The total duration of an experiment in milliseconds is thus given by the following formula:

$$duration = clocktick \times maxclock$$

Finally, tag `<controller_path>` contains a colon separated list of directories where con-

[1] At each clock tick the robot control steps are executed.
[2] Therefore, also the maximum number of control decisions.

troller libraries are compiled.

Tag <controllers> contains the list of controllers used in the experiment. As explained in Section 4.1, each controller is registered in the system with a user defined tag, so this portion of the XML is the place where these tags are used. Each controller is configured by specifying an identifier and the filename of the library containing the compiled code. Furthermore, inside each <controllers> portion, the list of the actuators/sensors used by the controller is reported, so that the simulator can initialize them properly. Each actuator/sensor is first identified by its type (for instance <footbot_proximity> or <pie_camera>) and then, inside its portion, the desired implementation is selected. The final part of the controller definition named <parameters> is left to define by the user, who can insert there any relevant parameter for the internal logic of the controller. If needed, it is possible to insert the same controller type more than once, with the condition that the specified identifier is unique for each instance. This is useful when the user, for example, wants to use the same controller with different values in the <parameter> tag, and assign to some robots an instance and to other robots other instances. An example:

```
<controllers>

  <footbot_sample_controller id="fc"
                             library="footbot_sample_controller">

    <actuators>
      <footbot_wheels>
        <implementation>
          dummy_footbot_wheels
        </implementation>
      </footbot_wheels>

      <footbot_gripper>
        <implementation>
          dummy_footbot_gripper
        </implementation>
      </footbot_gripper>
    </actuators>

    <sensors>
      <pie_camera>
        <implementation>
```

```
          generic_pie_camera
        </implementation>
      </pie_camera>

      <footbot_proximity>
        <implementation>
          generic_footbot_polynomial_proximity
        </implementation>
      </footbot_proximity>
    </sensors>

    <parameters>
      <min_distance>7</min_distance>
      <wheels_speed>10</wheels_speed>
    </parameters>

  </footbot_sample_controller>

  ...

</controllers>
```

The definition of the objects populating the simulated arena takes place in the section delimited by the tag `<arena>`. The tag possesses two required attributes: `size`, which specifies the size of the arena in meters along the main axes of the Swarmanoid Space[3], and `optimization`, which instructs the arena to store data to favor either 2D physics engines such as the kinematic one, or 3D. When only 2D engines are used, or when the majority of the robots is in 2D engines, the optimization value should be set to `2D`, while in the opposite case it should be set to `3D`. The rest of the `<arena>` part is filled with a list of entities. For every entity at least position and orientation are specified; some entities may have more parameters, as shown in the following listing:

```
<arena size="10,10,3" optimization="2D" >

  <footbot id="fb">
    <position>9, 9, 0</position>
```

[3]The arena is a cube.

```
    <orientation>0, 0, 45</orientation>
    <controller>fc</controller>
  </footbot>

  <eyebot id="eb">
    <position>5, 5, 1</position>
    <orientation>0, 0, 0</orientation>
    <controller>ec</controller>
  </eyebot>

  <block id="obstacle">
    <position>5, 5, .25</position>
    <orientation>0, 0, 0</orientation>
    <size>.5, .05, .5</size>
  </block>

  <cylindric_prey id="cp">
    <position>8, 8, 0</position>
    <orientation>0, 0, 0</orientation>
    <radius>0.10</radius>
    <height>0.10</height>
  </cylindric_prey>

    ...

</arena>
```

The specification of footbots and eyebots, for instance, involves the indication of the identifier of the controller to use for each of them. The obstacle object is defined through a block, a stretchable brick-like entity used to model walls, obstacles, and other similar objects. This entity needs the specification of the size along the main axes of the Swarmanoid Space similarly to the size of the arena. Finally, to insert a cylindric prey, that is a target grippable object, it necessary also to indicate its radius and height. As we will see later in this section when we will describe tag `<arena_physics>` that assigns entities to physics engines, the identifiers of the entities must be unique.

Physics engine are configured in a similar way to controllers and entities:

```
<engines>

  <dummy_engine id="ground">
    <subclock>1</subclock>
    <perpendicular_axis>z</perpendicular_axis>
    <distance>0</distance>
  </dummy_engine>

  <dummy_engine id="sky">
    <subclock>1</subclock>
    <perpendicular_axis>z</perpendicular_axis>
    <distance>1</distance>
  </dummy_engine>

</engines>
```

<dummy_engine> is the name of the kinematic physics engine inside the simulator, mainly due to its straighforward internal logic. In the above listing two kinematic engines are created. The first, the ground, is located on the x, y plane. On the other hand, the sky is parallel to the ground but translated one meter above it. The identifier is specified as usual, always with the constraint of uniqueness. Tag <subclock> is another optimization parameter that sets how many times the physics engine is called for each simulation clock tick. It is usually set to one, meaning that for each main clock tick the engine updates the physics status just once. Setting this value to something greater than one increases the accuracy of the physics engine in collision detection, although at the cost of a decrease in performance. <perpendicular_axis> can be set to x, y or z and, as the tag name suggests, indicates the axis to which the plane of the engine must be perpendicular. Likewise, tag <distance> fixes the signed distance between the plane and the origin.

As anticipated, tag <arena_physics> assigns entities to physics engines. An entity can be assigned to any number of physics engines, zero included. The mapping logic is deducible from the following example:

```
<arena_physics>

  <engine id="ground">
    <entity id="obstacle" />
    <entity id="fb" />
```

```
    <entity id="cp" />
  </engine>

  <engine id="sky">
    <entity id="eb" />
  </engine>

</arena_physics>
```

Finally, visualizations are defined in the last tag, exactly `<visualizations>`. More than one module can be inserted in it, as the following example shows:

```
<visualisations>

  <text_render id="text_world">
    <file>experiment_output.txt</file>
    <precision>2</precision>
  </text_render>

  <opengl_render id="mainwindow">
    <window_size>1024, 768</window_size>
    <window_title>Sample Experiment</window_title>
    <camera_view_XYZ_0>3.30811, 3.36071, 1.9</camera_view_XYZ_0>
    <camera_view_HPR_0>39, -34, 0</camera_view_HPR_0>
    <camera_view_XYZ_1>6.2875, 3.33551, 1.25</camera_view_XYZ_1>
    <camera_view_HPR_1>125, -21.5, 0</camera_view_HPR_1>
    <camera_view_XYZ_2>7.05332, 6.88148, 0.83</camera_view_XYZ_2>
    <camera_view_HPR_2>-155, 0, 0</camera_view_HPR_2>
    <use_textures>true</use_textures>
    <write_frames>false</write_frames>
    <frame_directory>Movies</frame_directory>
    <frame_filename>frame_</frame_filename>
  </opengl_render >

</visualisations>
```

In the above listing, two visualizations are specified: a basic text one, and the 3D OpenGL

graphical one. The first accepts as parameters the file name to write the data to and the number of desired decimal numbers. The second, `<opengl_render>`, sets the window size and title, three camera view points, toggles the use of surface textures to `true` and video frame storing to `false`, although a default location and the basename for frames is shown in tags `<frame_directory>` and `<frame_filename>`.

4.3 A Sample Experiment

In this final section of the chapter we present a sample experiment to show in practice how controllers are developed with the Swarmanoid Simulator.

The experiment involves an eyebot and three footbots. The aim of the collective task is to retrieve a target object, that, following the social insect metaphor, we will term *prey*. After the retrieval, the task is considered accomplished when the object is brought to an area termed *nest*.

Moreover, to make this experiment more compliant with the control objectives of the Swarmanoid Project illustrated in Chapter 1, we require the robots to cooperate by enriching the task as follows.

First of all, in the arena two preys of different colors are present. Both the footbot and the eyebot can perceive them, but only one of the two preys is the right one to pick. The choice of the prey to pick is performed at random by the eyebot at the beginning of the experiment. Therefore, the footbot, even if it is able to perceive both preys, is not able to solve the task alone and needs to communicate with the eyebot.

Furthermore, we want to pursue as much as possible the biological inspiration, avoiding an explicit exchange of structured information with WiFi or Bluetooth. Therefore, communication between the eyebot and the footbots is obtained only by means of visual information. More specifically, footbots inform the eyebot about their status by means of the colored LEDS they are equipped with, while the eyebot guides the footbots to the prey and to the nest using a laser beam that projects a colored spot on the ground, easily perceivable by the cameras of the footbots.

The setup of the experimental arena is depicted in Figure 4.1. We can recognize the eyebot, the three footbots, the two preys and the nest.

The robot controllers have been developed following the behavior based approach. The resulting behavior diagrams for the footbots and the eyebot are reported in Figure 4.2 and 4.3, respectively.

Initially, footbots are in *Search for Laser* behavior and the eyebot is in *Search for Prey* behavior: in other words, the eyebot looks for the chosen prey having the laser switched off, while the other robots randomly wander in the arena waiting for the laser to be turned on.

When the eyebot perceives the prey, it switches to *Go to Prey* behavior: the laser is turned

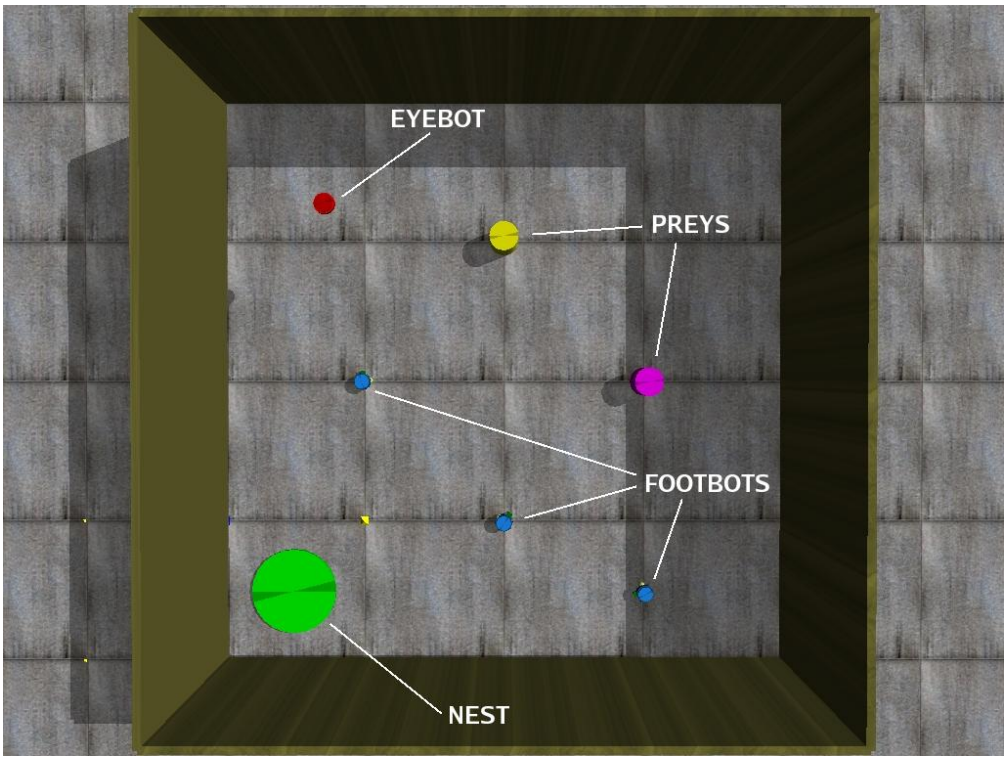


Figure 4.1: The initial setup of the experimental arena.

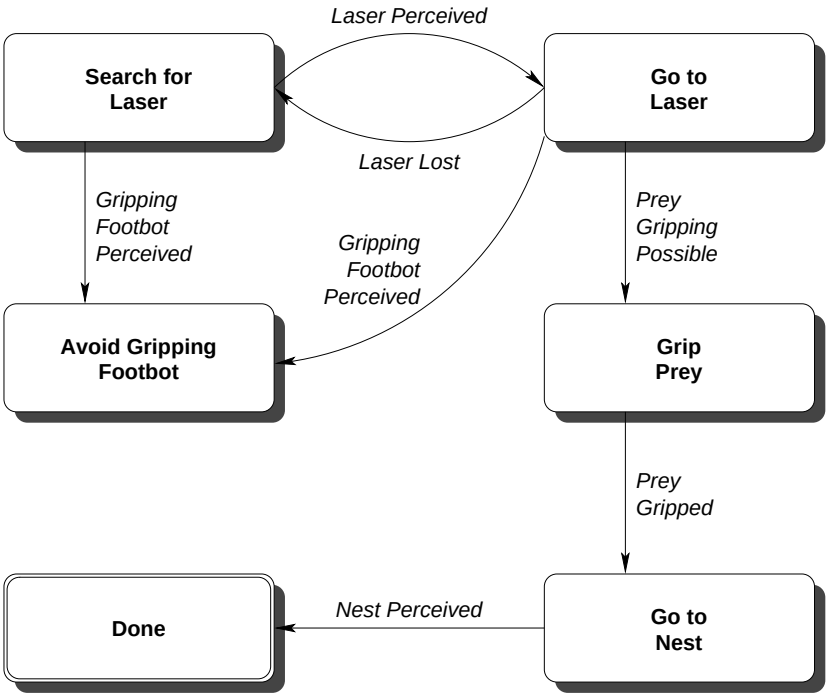


Figure 4.2: The behavior diagram of the footbot controller.

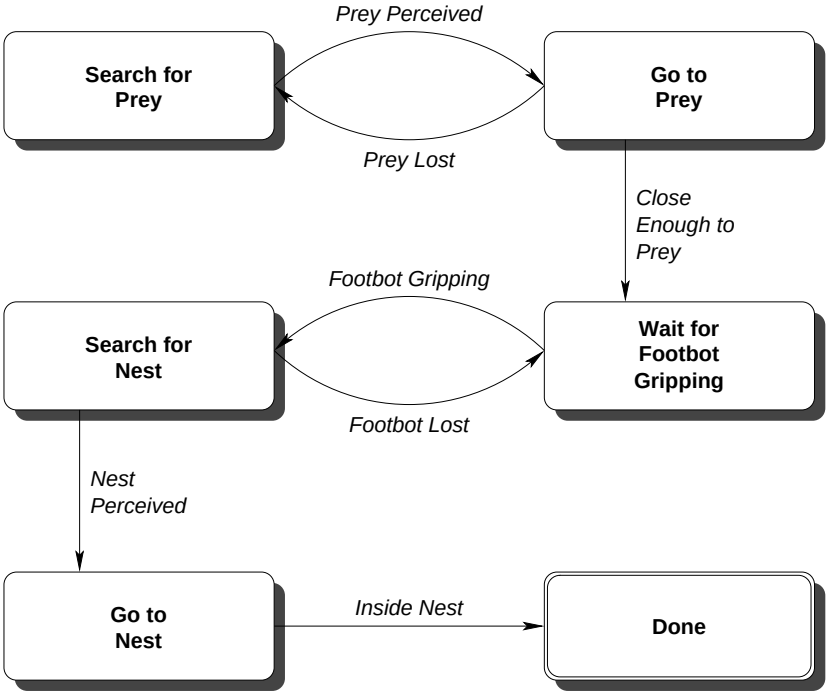


Figure 4.3: The behavior diagram of the eyebot controller.

on and the eyebot goes towards the prey. When close enough to it, the eyebot stops and waits for the footbots to arrive and grip the prey, switching to *Wait for Footbot Gripping* behavior.

In the meantime, footbots still search for the red colored spot on the ground, which is the projection of the laser on the ground. Eventually, one or more robots perceive it, thus changing their behavior to *Go to Laser*. Once a robot finally reaches the laser, it perceives also the nearby prey and approaches it, trying to grip it as soon as the distance is appropriate (*Grip Prey* behavior).

When a footbot successfully grips a prey, it changes its behavior to *Go to Nest* and sets its LEDs color to blue. This is a signal for the other footbots to switch to *Avoid Gripping Footbot* behavior, that is to stop trying to reach the laser or the prey and avoid the signaller as it carries the prey to the nest. The blue LEDs also inform the eyebot of the fact that finally a footbot is gripping the prey and that it should be brought to the nest. Therefore, when the eyebot perceives the blue lights of the footbot, it switches to behavior *Search for Nest*, to look for the green area. The footbot gripping the prey follows the laser. If the eyebot, for some reason, loses sight of the footbot, it simply stops, waiting for the footbot to arrive.

Eventually, the eyebot perceives the green area and drives the footbot there. When the prey is inside the nest, the task is accomplished. Figures 4.4-4.7 depict the highlights of the experiment.

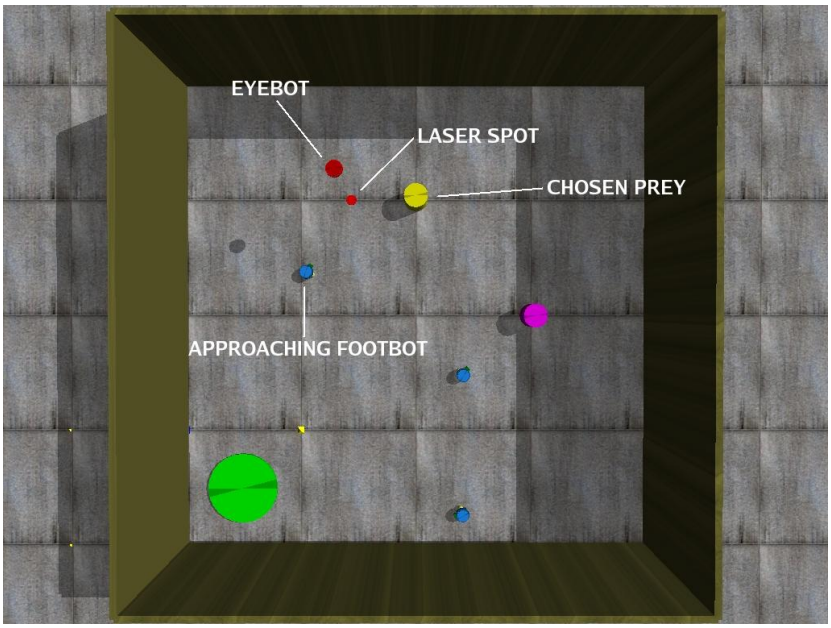


Figure 4.4: The eyebot waiting for a footbot to grip the prey.

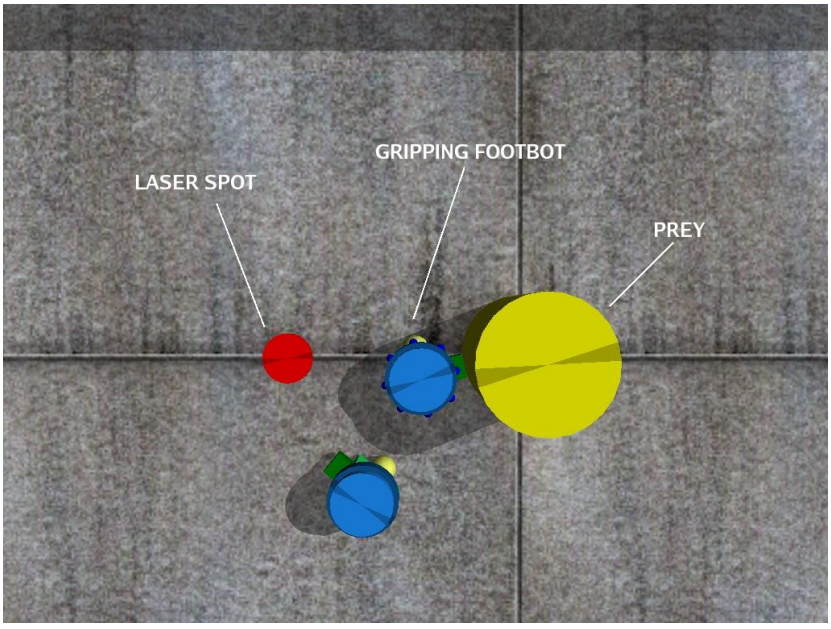


Figure 4.5: The footbot gripping the prey.

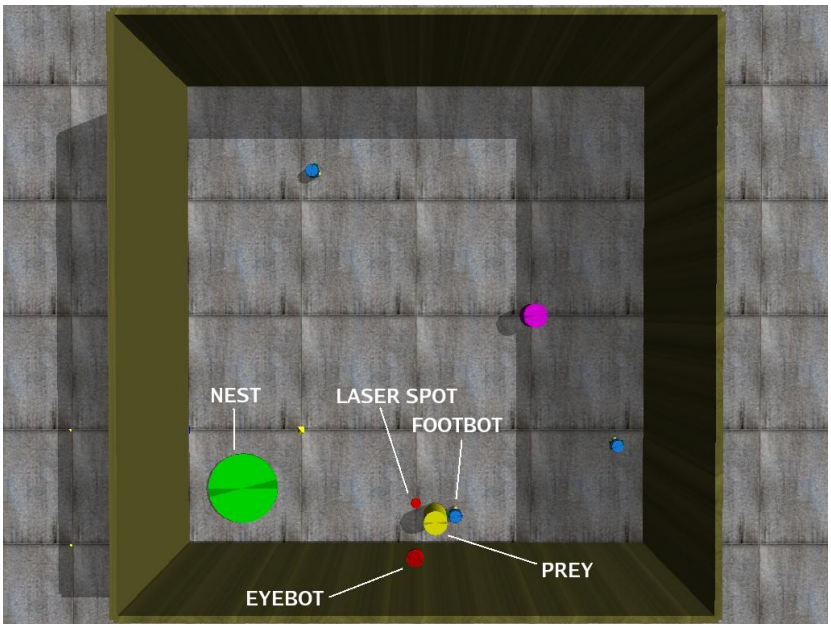


Figure 4.6: The eyebot driving the footbot to the nest.

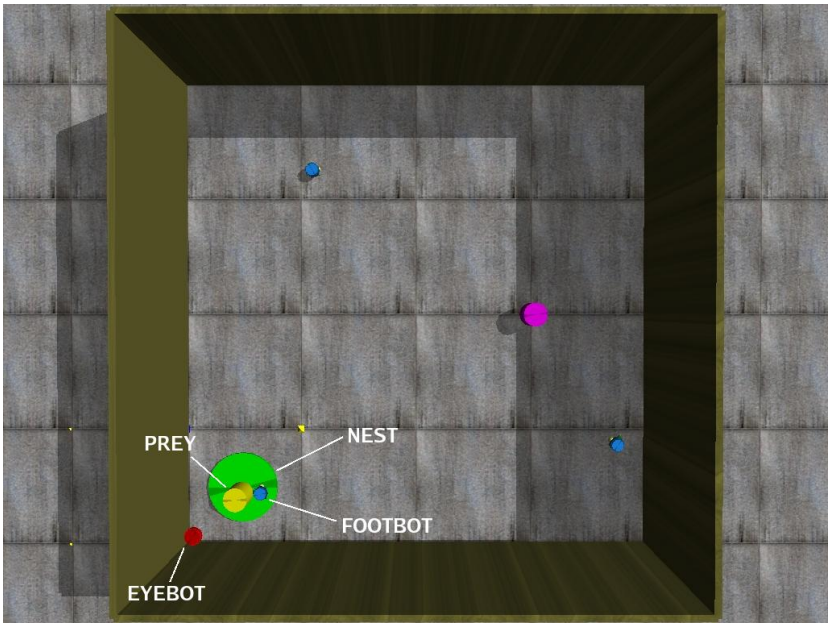


Figure 4.7: The prey is brought to the nest.

Chapter 5

Conclusions and Future Work

This final chapter summarizes the discussion of the previous chapters underlining the original contributions, and proposes possible improvements to the presented work.

Simulation studies play a central role in the development of robot prototypes as well as in the study of the properties of swarms, especially in terms of scalability and learning of control policies. The goals of the Swarmanoid Simulator consist in providing a software platform that:

- models the robots of the Swarmanoid and the environment they act upon according to multiple and selectable levels of detail, and provides for each level a faithful and consistent representation of the corresponding physical environment being simulated;
- is computationally efficient, to allow one to run simulations of complex real-world scenarios including a relatively large number of robots;
- allows an easy code migration from the simulator to the robot controllers to shorten implementation and debugging time;
- permits an efficient and comprehensive monitoring of the behaviors and performance of the robots through effective visualization and tracing interfaces;
- is highly modular, to facilitate independent code development from the different partners of the Swarmanoid Project;
- can be efficiently used for an effective evolution and evaluation of the different hardware alternatives and control policies that will be proposed during the project development.

In this document we have described the design choices that we have implemented to develop a software simulator that meets all the above requirements. Since none of the com-

mercial and free software simulators for multi-robot environments already available could effectively meet our needs, we opted for a full custom design and implementation.

The Swarmanoid Simulator has the following design characteristics. It is a continuous-time simulator specifically conceived to simulate the three different types of robots composing the Swarmanoid: eyebots, handbots and footbots, and their environment. These three types of robots fill the 3D space: eyebots either fly or stay attached to the ceiling, footbots move and act at the ground level, and handbots go upward from the ground and act at the level of the walls. Each robot is characterized by a physical structure, by a set of sensors and actuators, and by a controller module. The Swarmanoid Simulator provides multiple physics engines to cover different possibilities of modeling object interaction and movement in 2D and 3D representations of the physical world. A simulation run can include the concurrent use of one or more physics engines, with each engine devoted to model with the desired level of detail the Newtonian physics for the objects falling within a specific area or volume of the 3D environment space. The architecture is highly modular to allow the user to easily add and select actuators, sensors, physics engines, and renderers. The simulation scenario is configured using an intuitive XML syntax, while the most of the code is written in C++. The Swarmanoid Simulator design includes also multiple levels of graphical visualization based on the use of popular open source 3D graphical renderers such as 3D OpenGL and OGRE. A seamless transition between simulation and real robots is provided to the user since the interface for robot controllers is common to both simulated and real robots. In this way, a controller developed for a simulated robot needs only to be recompiled before it can be run on the real robot platform. A demostrative experiment has been presented to show the capabilities of the tool. Concurring with the goals of the Swarmanoid Project, the experiment involves a footbot and an eyebot that must cooperate to solve the task.

Future Work

Due to the high modularity and flexibility of its architecture, the Swarmanoid Simulator is constantly a work-in-progress. We plan to continue adding new modules as new research questions will require to adapt our tools.

At the time of writing, two physics engines are under study. First of all, we are designing a dynamic 2D physics engine that will extend the capabilities of the kinematic engine with full support of friction and rigid body dynamics. This engine will prove useful to model footbots connecting to each other or to handbots on the ground.

On the other hand, to model accurately eyebots and handbots, we are also implementing a 3D dynamic engine based on ODE.

The improvement of sensors and actuators is planned as soon as the robotic platforms are available. Sampling techniques will be employed when possible to increase realism and to save computational power.

For what concerns the architecture itself, we are considering the hypothesis of allowing robots to change the physics engine in charge of updating their status to another at run-time. This addition would not require any major modification to what we have already implemented and would allow to fully exploit the optimization opportunities given by the possibility to run concurrently different engines with different physics accuracies. The scenario in which this improvement will be more useful is the transport of a handbot by a swarm of footbots to a target area, where the handbot shoots the rope, climbs the wall and retrieves an object. In the first phase, the transport, the handbot is idle, therefore a precise modeling of it is not required: a 2D physics engine is largely sufficient. On the other hand, in the second phase, the handbot dynamics require a three-dimensional engine. The proposed enhancement of the architecture would solve the problem in the best possible way.

Bibliography

Arkin, R. (1998). *Behavior-Based Robotics*. Intelligent Robots and Autonomous Agents. MIT Press, Cambridge, MA, USA.

Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Science of Complexity. Oxford University Press, New York, NY, USA.

Camazine, S., Deneubourg, J.-L., Franks, N. R., Sneyd, J., Theraulaz, G., and Bonabeau, E. (2003). *Self-Organization in Biological Systems*. Princeton Studies in Complexity. Princeton University Press, Princeton, NJ, USA.

Carpin, S., Lewis, M., Wang, J., Balakirsky, S., and Scrapper, C. (2007). USARSim: a robot simulator for research and education. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1400–1405.

Christensen, A. (2005). Efficient neuro-evolution of hole-avoidance and phototaxis for a swarm-bot. Diplôme d'Etudes Approfondies en Sciences Appliquées thesis, IRIDIA, Université Libre de Bruxelles.

Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T. H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D., and Gambardella, L. M. (2004). Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2–3):223–245.

Frigg, R. and Hartmann, S. (Spring 2006). Models in science. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*.

Gerkey, B., Vaughan, R., and Howard, A. (2003). The Player/Stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th Int. Conf. on Advanced Robotics (ICAR 2003)*.

Gillies, D. (1993). *Philosophy of Science in the Twentieth Century*. Blackwell, Oxford, UK.

Go, J., Browning, B., and Veloso, M. (2004). Accurate and flexible simulation for dynamic, vision-centric robots. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'04)*.

Grassé, P.-P. (1959). La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–83.

Jacobi, N. (1997). Half-baked, ad-hoc and noisy: Minimal simulations for evolutionary robotics. In Husbands, P. and Harvey, I., editors, *Proceedings of the Fourth European Conference on Artificial Life: ECAL97*, pages 348–357. MIT Press, Cambridge, MA, USA.

Klein, J. (2002). Breve: a 3D simulation environment for the simulation of decentralized systems and artificial life. In *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*.

Kramer, J. and Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132.

Ljung, L. (1999). *System Identification*, chapter 1, page 6. Information and System Sciences. Prentice Hall, second edition.

Menon, C., Murphy, M., and Sitti, M. (2004). Gecko inspired surface climbing robots. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO'04)*.

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

Mondada, F., Pettinaro, G. C., Guignard, A., Kwee, I. W., Floreano, D., Deneubourg, J.-L., Nolfi, S., Gambardella, L. M., and Dorigo, M. (2004).

Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics*. Intelligent Robots and Autonomous Agents. MIT Press, Cambridge, MA, USA.

Salomon, D. (2004). *Data Compression: The Complete Reference*. Computers / General Information. Springer.

Seugling, A. and Rölin, M. (2006). Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. Master's thesis, Department of Computer Science, Umeoa University, Sweden.