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

Simulation of Handbot Controllers

The implementation of a novel 3D simulation platform

Javier Martinez Gonzalez

Promoteur 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 2007/2008

Abstract

This work deals with the design and the implementation of a 3D simulation platform for the swarmanoid project, specifically the 3D dynamic engine and the 3D visualisation. Validation has been achieved through an experiment beeing simulated on the platform. The experiment requires de collaboration of two types of robots: Hand-bots, manipulation robots, and Foot-bots, wheel based robots. In this experiment, hand-bots need to lift an heavy object but the initial geometry does not allow them to achieve this. Foot-bots carry Hand-bots allowing them to reach a more suitable geometry to finally lift the heavy object.

Acknowledgments

I would first of all like to thank Prof. Marco Dorigo for his supervision and for the opportunity I had to be a part of the Swarmanoid project for two years. Working at IRIDIA was an everyday pleasure, as it was an extremely friendly and stimulating research environment.

I also would like to thank Rehan and Mauro for their comments and reviews throughout my stay at the lab. I specially thank Mauro for the review of this work.

I would like to thank Carlo, Alvaro, Frederic, Giovanni and Vito for all the effort made to provide such a nice simulator as the one we have. It is in some way our baby.

I thank all the people i met through IRIDIA. In a random way, they are: Hugues, Marco, Muriel, Mauro, Elio, Bruno, Francisco, Thomas, Christos, Prasanna, Saifullah, Arne, Alex, Eliseo, Renaud, Max, Amin, Marco, Shervin, Rehan, Carlo, Giovanni, Took, Eric, Nithin the coffee swiss guard, Giacomo, Carlotta, Christophe, Vito, Francesco, Stephane, Michael and Alvaro.

Finally, I also would like to thank Jorge Cham and Sacha Baron Cohen for the everyday zygomatic exercise.

Contents

Abstract iii

Acknowledgments v

Contents vii

List of Figures x

1 Introduction 1
- 1.1 Swarm robotics . . . 2
- 1.2 The Swarmanoid project . . . 3
  - 1.2.1 Project hardware . . . 3
  - 1.2.2 Project goals . . . 4
- 1.3 Problem statement . . . 4
- 1.4 Contributions . . . 5
- 1.5 Outline . . . 5

2 Simulation 7
- 2.1 The choice of a custom design . . . 7
- 2.2 Argos . . . 8
  - 2.2.1 The architecture . . . 8
  - 2.2.2 The flow of execution . . . 10
  - 2.2.3 Modules . . . 11

3 Visualisation 13
- 3.1 Real time visualisation . . . 13
- 3.2 Render library choice . . . 14
  - 3.2.1 Ogre features . . . 15
- 3.3 Plugin design . . . 16
  - 3.3.1 Code and data structure . . . 17
- 3.4 Swarmanoid entities . . . 18

3.4.1 3D skills . . . 19
3.5 GUI . . . 20
3.6 Features . . . 22

4 Physics **29**
4.1 The state of art . . . 30
4.2 3D dynamics plugin design . . . 31

5 Simulated sensors and actuators **33**
5.1 Foot-bot sensors and actuators . . . 33
5.1.1 Mobility . . . 34
5.1.2 Proximity sensors . . . 34
5.1.3 Gyroscopic sensor . . . 35
5.1.4 Docking module . . . 35
5.1.5 Rotating distance sensor . . . 36
5.1.6 Omnidirectional camera sensor . . . 36
5.1.7 Communication module . . . 37
5.2 Hand-bot sensors and actuators . . . 37
5.2.1 The hull . . . 37
5.2.2 Rope launcher . . . 37
5.2.3 Arms and head . . . 38
5.2.4 Grippers . . . 39
5.3 Eye-bot sensors and actuators . . . 41
5.3.1 Propulsion actuator . . . 41
5.3.2 Vision sensor and laser actuator . . . 41

6 Simulated experiments **43**
6.1 Scenario . . . 43
6.1.1 Motivation . . . 43
6.1.2 Overview . . . 43
6.1.3 Methodology . . . 44
6.2 The Push-Pull controller . . . 44
6.2.1 Foot-bot control . . . 45
6.2.2 Hand-bot control . . . 46
6.2.3 Results . . . 46
6.3 Reusable behaviour design . . . 48

7 Conclusions **51**

Appendix **55**
.1 3D object loading 55
.2 3D module architecture 61
.3 Actuators and sensors 64

List of Figures

1.1 Example of swarm collaboration. 2
1.2 hardware design. 3

2.1 Overall architecture of the Swarmanoid simulator. 9

3.1 The Argos render interface. 16
3.2 Module file structure. The small rectangular coloured boxes represent the proportional size of each folder compared to the whole module. 17
3.3 The building process of the hand-bot mesh. 18
3.4 Results of the 3D CAD model conversion. 19
3.5 Real-time hand-bot models rendered in the simulator. 20
3.6 Module GUI modes. 21
3.7 GUI Modules. 21
3.8 Module console. 22
3.9 View of robots states. 23
3.10 Moving a hand-bot while an experiment runs. 24
3.11 Texture management of entities. 25
3.12 Camera travelling. 26
3.13 Module special effects available. 26
3.14 Visual elements added to the simulation. 27

4.1 Example of available joints in ODE. 31

5.1 Modules of a foot-bot. 33
5.2 The treel driving system (left). Details of the wheel (right). 34
5.3 The current prototype of the foot-bot sensors board (left). Ground proximity sensor (center). Contact ground sensor (right). 35
5.4 Gyroscopic sensors help keeping track of the current rate of acceleration on each axis. 35
5.5 Docking module of a foot-bot. 36
5.6 The foot-bot rotating distance sensor. 36

5.7 The foot-bot camera sensor. 37
5.8 The possible movements of the hand-bot. 38
5.9 View of a hand-bot. 39
5.10 The rope launcher of the hand-bot. 40
5.11 The gripper of the hand-bot. 41
5.12 Current quad rotor platform: A.) collision protection ring, B.) brushless motor, C.) contra-rotating propellers, D.) LIPO battery, E.) high-speed motor controller, F.) flight computer, G.) infrared distance sensors and ceiling attachment. 42
5.13 Omni-directional vision system: left - prototype, right - mounting location. . 42

6.1 Full scenario trial with 4 hand-bots and 14 foot-bots. 44
6.2 A foot-bot connection to a hand-bot at a precise location (green-blue slot). . . 45
6.3 The Push-Pull finite state machines. 46
6.4 Interaction between a hand-bot and two foot-bots. 47
6.5 Successful experiment with two hand-bots and three foot-bots. 47
6.6 Obtained data from a Push-Pull with two hand-bots and different foot-bots numbers. 48
6.7 An OO state pattern controller. 50

1 The Rendering SubClasses present inside Argos. 62
2 The COgreRender collaboration diagram. 63
3 Unimplemented eye-bot sensors. 64

Chapter 1

Introduction

Artificial intelligence. What a wonderful field. Quite complex when you start to think about it. Some textbooks simply describe it as the study and design of intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions. Although concise as a definition, it holds the hopes that many scientists (such as myself) have, that one day machines will exhibit reasoning, knowledge, planning, learning, communication, perception, social skill, creativity and the ability to move and manipulate their environment. Will that day come? No one knows. Despite the uncertainty, artificial intelligence (AI) has become a huge multi-disciplinary field, gathering more and more minds around the deep questions underlying its core.

Just as many other scientific fields, artificial intelligence tries to study and describe phenomenas and processes that take place in its own field. Hypothesis get formulated from proposed theories, experiments get done to validate or infirm sustained ideas, models are described to state observed, or to be observed, behaviours. But AI holds something particular of its own, something really special. In the mythology, the Ourobouro's snake is represented by a circle: he is eating his own tail, and symbolises all problems where the goal is one of the variables. AI posses in its own definition an amusing paradox: it studies itself. Indeed, as we could define its field of study as the understandings of the mechanics of the comprehension. Of course other disciplines have the same ambition, cognitive sciences, psychology, linguistics, neurosciences, etc. But only AI does not verify its theories and models on humans. The verifying and investigation tool is a *universal machine*[1].

Computational intelligence, a branch of AI, sees a Turing machine as a tool and a goal. It tries to create programs that are, in some sense, intelligent, by combining elements of learning, adaptation, evolution and logic. Inside the boiling topic of AI, robotics has become the framework to many of theses experiments. Within computational intelligence we find a sub branch called *Swarm robotics*.

[1] A Turing machine that is able to simulate any other Turing machine is called a Universal Turing machine (UTM), or simply a universal machine.

1.1 Swarm robotics

Swarm robotics (Bonabeau et al., 1999) is the study of how large numbers of relatively simple physically embodied agents can be designed such that a desired collective behaviour emerges from the local interactions among agents and between the agents and the environment.

Swarm robotics is a novel approach to the coordination of large numbers of robots. Relatively close to minimalist robotics, it is inspired by the emergent behaviour observed in social insects. Social insects are known to coordinate their actions to accomplish tasks that are beyond the capabilities of a single individual (Camazine et al., 2003): ants foraging, termites building huge mounds, schools of fishes avoiding predators. Such coordination capabilities are still beyond the reach of current multi-robot systems.

In the last fifteen years researches have been conducted to understand these social behaviours, how these populations evolve, interact and accomplish tasks. As seen in biological life, relatively simple individual rules can produce a large set of complex behaviours. Unlike distributed robotic systems in general, swarm robotics emphasizes on having a large number of robots, and promotes scalability, for instance by using only local communication. The leitmotives of swarm robotics are flexibility, robustness, decentralization and self-organization.

(a) Weaver ants stitching a leave to make a nest.

(b) The s-bot mobile robot climbing a step in the swarm-bot configuration. See (Mondada et al., 2004) for more information.

(c) School of the colorful squirrelfish Sargocentron xantherythrum. Useful for predator avoidance.

Figure 1.1: Example of swarm collaboration.

1.2 The Swarmanoid project

Following the ground bases of swarm robotics, the *Swarmanoid project* is a future and emerging technologies (OPEN-FET) project within the framework of swarm robotics, funded by the European Commission, that temporally and logically follows the Swarm-bots project (Dorigo et al., 2004).

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.

1.2.1 Project hardware

The Swarmanoid is comprised of a relatively large number of autonomous robots of three types:

Foot-bots Are specialised in moving on rough terrain and transporting either objects or other robots; they are based on the robotic platform developed within the European Swarm-bots project, the S-Bot (Mondada et al., 2004).

Hand-bots Are specialised in moving and acting in a space zone between the one covered by the foot-bots (the ground) and the one covered by the eye-bots (the ceiling). Hand-bots can climb vertical surfaces of walls or objects located in the environment. The hand-bot is the most innovative robot of the Swarmanoid project.

Eye-bots Are specialised in sensing and analysing the environment from a high position to provide an overview that foot-bots or hand-bots cannot have. Eye-bots fly or are attached to the ceiling.

(a) A foot-bot. (b) An eye-bot. (c) A hand-bot.

Figure 1.2: Robots present in the Swarmanoid project.

A closer description of the robots will be given throughout the Chapters 4 and 5.

1.2.2 Project goals

The ultimate goal of the Swarmanoid project 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[2] approach. In the traditional methodology, first a controller for a single robot is developed, then swarm behaviour with similar robots is inserted, and then finally interaction with the other types of robots is added. The project avoids as much as possible this approach, favouring the opposite one, the development of controllers in parallel.

1.3 Problem statement

Developping prototypes and controllers for the Swarmanoid project is far from trivial. Simulation studies do play a major role in the development of the prototypes of the robots as well as in the study of the properties of the swarmanoid, especially in terms of scalability and learning of control policies. Due to the heterogeneous expertise of the researchers working in the project, the simulator platform 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 a behavior based (Arkin, 1998) approach or a evolutionary techniques (Nolfi and Floreano, 2000) approach. Therefore, a convenient simulation framework to pursue the control goals of the Swarmanoid project was developped: *Argos* (see chap 2).

Argos is a novel multi-engine and multi-robot architecture of a three-dimensional physics simulator. Argos let's the user choose between different types of visualisations and physics engines. Each engine and visualisation is seen as an external plugin, thus allowing multiple instances of a plugin to live within the simulated experiment. By three-dimensional physics simulator, we state that the simulated world has 3D coordinates, but the corresponding physical mappings can be achieved using various techniques, such as the use of many 2D kinematic[3] engines. While for some experiments, a 2D kinematic engine coupled with a simple visualisation might be more than satisfactory, complex munipulations such as the one's accomplished by the hand-bots need a more accurante and realalistic simulation framework. Indeed, validating a climbing or object grasping model in 2D is not possible, therefore the need for some more appropriated tools like a 3D dynamic[4] engine and a 3D visualisation

[2]Holism (from holos, a Greek word meaning all, entire, total) is the idea that all the properties of a given system (biological, chemical, social, economic, mental, linguistic, etc.) cannot be determined or explained by its component parts alone. Instead, the system as a whole determines in an important way how the parts behave.

[3]Kinematics (Greek , kinein, to move) is a branch of dynamics which describes the motion of objects without consideration of the circumstances leading to the motion. In other words, it only deals with the geometric aspect of motion.

[4]Dynamics is the branch of classical mechanics that is concerned with the motion of bodies. It is divided into

where needed.

1.4 Contributions

This work deals with the implementation of a 3D simulation platform for the swarmanoid project, specifically the 3D dynamic physical plugin, the 3D visualisation plugin and the subsequent sensors and actuators for the concerned robots (mostly foot-bots and hand-bot).

Validation is achieved through an experiment beeing simulated on the platform. The experiment requires de collaboration of two types of robots: hand-bots and foot-bots. In this experiment, the hand-bots need to lift an heavy object but initial geometry does not allow them to achieve this. Foot-bots carry hand-bots allowing them to reach a more suitable geometry to finally lift the heavy object. The obtained controller uses a novel mechanism that we call the Push-Pull mechanism.

A general code structure for controller behaviours reuse is also presented.

1.5 Outline

In the following chapters, we present the resulting implementation of the Swarmanoid simulator and the validating experiment. Chapter 2 give a better view of Argos, the swarmanoid simulator. Subsequently, the discussion of Chapter 3 and 4 we describe the 3D physical engine and the 3D visualisation tool. Chapter 5 and 6 presents the simulator from the point of view of the actuators and sensors. We illustrate the software design that models the hardware embedded within the robots. Chapter 7 demostrate an experiment involving foot-bot and hand-bots. The cooperation of the two robots is required to lift an heavy object. The Push-Pull mechanism is explained in details and a controller behaviour reuse methodology is extracted. Chapter 8 summarizes the main achievements of the presented work and outlines future directions for its improvement.

two branches called kinematics and kinetics. Kinetics is concerned with the motion of bodies under the action of given forces.

Chapter 2

Simulation

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 embedding environments. On the downside, the intrinsic complexity of a (multi-)robot system and of the real-world environment it is supposed to act upon sometimes makes it hard to design realistic simulation models that allow deriving sound evaluations and predictions of the robotic system at hand.

Many simulators for robotics already exist. A nice review on development tools for multi-robot systems can be found in the 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.

But the Swarmanoid project as described in Chapter 1 presents many challenges for the development of a simulator. A fully custom design was chosen, instead of using as reference platform one of the available simulators, such as Gazebo (Gerkey et al., 2003), USARSim (Carpin et al., 2007), or Webots (Michel, 2004). In the following chapter we describe the motivations supporting this choice and briefly make a short description of the architecture of the Argos simulator.

2.1 The choice of a custom design

If we want to simulate the specific characteristics of the robots composing the Swarmanoid we need to re-implement from scratch the majority of the sensors and actuators. For instance, none of the existing simulators include modules that could help to simulate the hand-bot climbing along the vertical dimension by shooting a rope that gets magnetically attached to the ceiling. Additionally, relying on third parties libraries is a good choice, but relying on general software platforms selected by a third party is not a good one, especially if you have to heavily modify it. All current robot simulators do not suit well the swarm robotics philosophy, which demands lots of entities to be simulated at the same time. However, the

Breve simulator developed for the I-SWARM (Seyfried et al., 2005) project, despite some promising aspects, lacks the potential for complex controllers as it uses a scripting language.

Another critical issue for a simulator is the computational performance. In the case of the Swarmanoid project, running times play a critical role for the usability of the simulator as we have to deal with multiple highly heterogeneous robots interacting in realistic abstractions of real-world scenarios, and from the other hand, we use computationally-demanding evolutionary algorithms to synthesize robot controllers. This infers a strong need to allows 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 eye-bot is marginal, such that it does not make much difference whether their behaviour is simulated with high physical accuracy or not, physics of eye-bots at the ceiling level can be managed by a simplified engine while a more accurate physics engine can be used for the foot-bots, optimizing in this way the use of computational resources without losing relevant information.

To deal with all these issues, Argos, the Swarmanoid simulator was created.

2.2 Argos

2.2.1 The architecture

The architecture of the Swarmanoid simulator follows a well structured design as seen in a schematic representation is given in Figure 2.1.

As can be seen from the Figure, the simulator architecture is organized around one single component, the *Swarmanoid Space*. This is a central reference system representing the current state of the simulation at each simulation step: it contains information about the position and orientation of each of the simulated entities, including the robots of the Swarmanoid and all other objects that are present in the simulated environment. The other components of the simulator interact mainly with the Swarmanoid Space for their working. Physics engines take the current state of the Swarmanoid Space as their input, calculate physical movements and interactions based on the actions of the different simulated entities, and then update the Swarmanoid Space again with the new state of the simulated system. Renderer's visualize the content of the Swarmanoid Space at each simulation step. Sensors and actuators can interact either with the Swarmanoid Space or directly with the physics engines. The choice between these two options depends on the needs of each individual sensor or actuator. Sensors that require the calculation of physics equations for their input, or actuators that can have physical effects on their environment, need to interact with the physics engines for their operations. They are called specific sensors/actuators, since they need a specific implementation for each physics engine that is used. All other sensors and actuators that interact only with the Swarmanoid Space are called generic sensors/actuators. Finally, robot

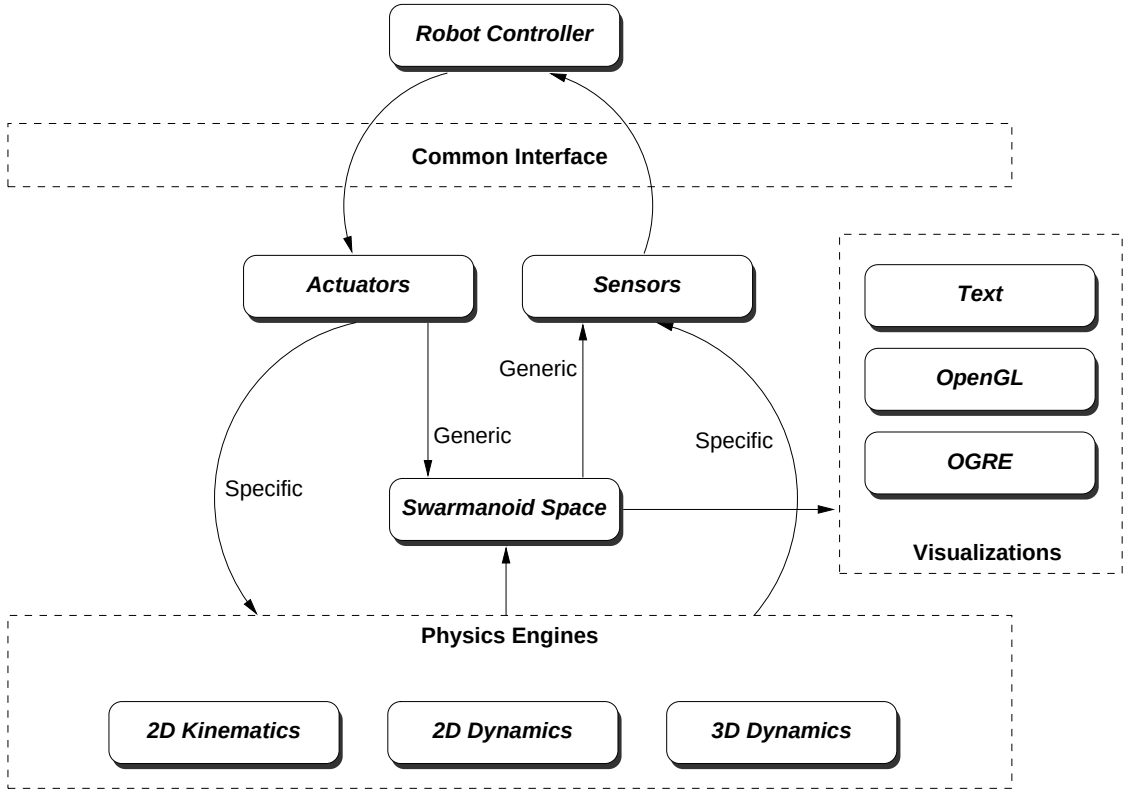


Figure 2.1: Overall architecture of the Swarmanoid simulator.

controllers interact with the rest of the simulator only through the sensors and actuators, as is the case for controllers that are placed on real robots.

An important aspect of the chosen architecture is the high degree of modularity. Thanks to the use of the Swarmanoid Space as a central reference system, all other components of the simulator can be relatively independent. Each actuator, sensor, renderer, physics engine or robot controller can therefore be considered as a separate module, that can be plugged in to the system relatively easily. It is sufficient that each module implements the interfaces that are present in the simulator for its type of component, and that it calls the relevant registration code (a separate auto-registration macro has been developed for each type of component). This modular system has important advantages. A first advantage is that it facilitates the cooperation between different developers that work at various sites on different parts of the software. A second advantage is that it allows to develop a large number of different modules and implementations and gives the user the choice of which of these to use for each experiment. In particular, the current prototype of the Swarmanoid simulator contains already a variety of physics engines and renderers, and different implementations of various sensors and actuators. Each of these modules and implementations has its advantages over others of the same type. For example, a two-dimensional kinematics physics engine will give less precise results than a three-dimensional dynamics physics engine, but

might be more computationally efficient. The user can, for each component of the simulator, choose the implementation that fits his needs best, and can even use different implementations simultaneously in the same experiment.

Another important property of the simulator architecture we want to point out here is the use of the *common interface*. This is a collection of interfaces that define the functions that are available to a robot controller to interact with the sensors and actuators. The common interface will be the same on the real robots as it is in the Swarmanoid simulator. This should greatly improve the portability of controller code from the simulator to the real robots. Inside the simulator, each sensor or actuator interface can have multiple implementations, according to the modular approach described above, giving users the possibility to choose the implementation that fits their experiment best.

2.2.2 The flow of execution

The flow of execution during a run of the Swarmanoid simulator is as shown below. It is given so that the reader can understand how the simulation internal loop works.

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 visualization*)

2-1 Display the status of the Swarmanoid Space.

3. Pre-physics control

3-1 Execute user defined pre-physics loop function.

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

4-1 Each robot controller is executed. Internally, a controller reads the sensor inputs and, according to the control logic, outputs values to the actuators.

4-2 The physics engine reads the status in the Swarmanoid Space of the objects assigned to it, and the outputs of their actuators.

4-3 Then, it updates position and heading of the assigned objects and resolves local collisions.

4-4 Finally, it updates the status of the Swarmanoid Space.

5. Post-physics control

5-1 Execute user defined post-physics loop function.

6. Next Simulation Step

6-1 If Time Limit is reached or Stopping Conditions are met, end 'the simulation.

6-2 Else Return to point 2.

2.2.3 Modules

The procedure to add new modules is similar for all modules inside Argos. Inserting a new module in the simulator is as simple as creating the classes that implement the relevant interfaces. Each module type is therefore essentially an implementation of an interface which is then registered in the system thanks to the already mentioned auto-registering mechanism.

The actual implementation of the mechanism is transparent to the user. It basically relies on a memory structure created at run-time. The memory structure consists of a set of maps, one for each family of module: controllers, loop functions, physics engines, renderers, actuators and sensors. Each family of module contains different types of module: for example, the actuator family contains as types the foot-bot wheels actuator, the eye-bot propeller actuator; the renderer family contains the types text, OpenGL[1] and OGRE[2], and so on. Each family map therefore links types to their class constructor: (module-type, module-constructor). Calling the class constructor makes it possible to create an instance of the wanted module. The contributor registers a new module calling a macro. The macro code is run when the operating system dynamically links of the needed libraries: therefore the modules are available even before the first instruction of the simulator is called. When the simulator starts, all the modules are already linked and ready to be used.

[1]OpenGL (Open Graphics Library) is a standard specification defining a cross-language cross-platform API for writing applications that produce 2D and 3D computer graphics. `http://www.opengl.org/`

[2]OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D engine written in C++.

Chapter 3

Visualisation

Chapter 2 gave us a general view of the Argos simulator.

This chapter describes the existing visualisation tools and shows the motivations that pushed us to develop a new and more advanced 3D visualisation module. The discussion continues with a first overall view of the way the module is divided into modules and of the mutual connections among them, detailing the internals of each module.

3.1 Real time visualisation

When a virtual experiment is held in robotics, we are strongly interested in the data it produces, such as the adequacy of each individual robot to deal with the current task, the accuracy of the controller, etc. The problem, is that getting this data, requires a tedious work of controller creation, enhancement and fine tuning. To obtain such controller, sometimes computing an objective function to quantify the optimality of the controller is enough, but most of the time, we need to actually see what is really happening in the simulated world. Therefore a rendering is needed. Two types of rendering can occur: Real-time and Offline.

Offline rendering is the final process of creating the actual 2D image or animation from a finished experiment. Several different, and often specialized, rendering methods are available. This range from the distinctly non-realistic wire frame rendering through polygon-based rendering, to more advanced techniques such as: scanline rendering, ray tracing, or radiosity. While this gives a very nice output, controllers developers cannot cope with this as most of the offline rendering techniques need a huge amount of power and rendering may take from seconds to days for a single image (frame).

On the other hand, real-time rendering techniques let us see directly the state of the simulated world without having to wait some later calculation, thus allowing us a faster controller development cycle. Also, thanks to the capabilities of today's graphical cards, computing power has finally evolved to the point where its feasible to render something as complex as the human face interactively. As a result, real-time graphics in the last few

years had sort of a renaissance, treating our eyes to a new kind of visuals that would have been unthinkable in real time even five years ago. One thing that we discovered is that the level of sophistication that perhaps attracted us to the subject initially also presents itself as a near-vertical learning curve. Getting all those nice graphical effects into an application is hard.

Sure, libraries like OpenGL are handy, but after playing with these for a while, it did not take long before it was realised just how much additional work there was to do before we could lay a practical framework to really start being productive beyond simple demos.

Therefore, the choice was made to use an already build real-time rendering API to achieve our advanced 3D rendering module for Argos.

3.2 Render library choice

Three criterias were used to shortlist possible render libraries. The first one was the fact that we needed a license allowing its use as a module inside Argos. Another issue was the fact to not restrict the simulator to a platform; therefore it had to be able to run on multi-platforms (Linux, MacOs and windows). A third criterion was that we wanted something used in real applications, thus stable and mature. The possible restricted choices are listed in 3.1.

Table 3.1: Some of the most popular rendering engines with their web address.

Crystal Space	`http://www.crystalspace3D.org`
Delta3D	`http://www.delta3D.org`
Irrlicht	`http://irrlicht.sourceforge.net/`
Panda3D	`http://www.panda3D.org`
Ogre	`http://www.ogre3D.org/`
OpenSceneGraph	`http://www.openscenegraph.org/`
V3X 3D	`http://www.realtech-vr.com/v3x/`

First thing we wanted was the ability to easily extend the library with a GUI[1]. This left us with four possible choices: Crystal Space, Irrlicht, Panda3D and Ogre. Second, mesh loading, thus the ability to load complex 3D model should be easy and standard. This ruled out Panda3D. For the rest, all remaining engines had strong similarities. They all were written in C/C++, they were of an objected-oriented design, capable or using Lua[2] scripting language, using shadow mappings, all possible lightning types, having level of detail rendering with occlusion culling, etc. The final choice was then decided on the documentation of the library. Ogre was the clear winner.

[1] A graphical user interface (GUI).

[2] The Lua programming language is a lightweight, reflective, imperative and procedural language. Mainly used for AI and fast rule engine in game logic.

3.2.1 Ogre features

Ogre features match, and in some cases surpass, nearly every capability offered by a commercial 3D rendering package:

- Full and equal support for both OpenGL and Direct3D
- Full support for Windows, Linux, and Mac OS X platforms
- Library is threadable, thus allowing rendering to be spanned across different CPU's
- Simple and extensible object framework, easily integrated into an existing application
- Automatic handling of render state management and hierarchical culling
- Powerful and sophisticated material management and scripting system, allowing maintenance of materials and fallback techniques without touching a single line of code
- Support for all fixed-function texture and blending techniques, as well as programmable GPU techniques and all high-level and assembled shading languages, such as Cg, HLSL, and GLSL
- Support for a wide variety of image and texture formats including PNG, TGA, DDS, TIF, GIF, JPG, as well as odd formats such as 1D, volumetric textures, cube maps, and compressed textures such as DXTC
- Full support for render-to-texture techniques and projective texturing (decals)
- Full support for material LoD (level of detail, mipmapping) techniques
- Optimized binary mesh format with both manual and automatic LoD generation
- Official and community support and development of exporters from all major commercial and open source 3D modelling packages to the Ogre mesh and animation format
- Full access to vertex and index buffers, vertex declarations, and buffer mappings
- Full support for skeletal and pose (vertex) animations, as well as sophisticated blending of any number of each and multiple bone weights per vertex
- Support for both software- and hardware-accelerated skinning
- Support for static geometry batching
- Support for biquadric Bezier patches
- Plug inbased hierarchical scene structure interface, allowing you to use the scene graph that best suits your application (basic octree scene manager included as an example module)
- Advanced maskable scene-querying system
- Full support for several shadowing techniques, including stencil, texture, additive, and modulative, all with full support for hardware acceleration
- Advanced plug inbased particle system with support for extensible emitters, affectors, and renderers (sample ParticleFX plug-in included)
- Full support for easy-to-use skyboxes, skyplanes, and skydomes
- Billboarding for sprite-based graphics and rendering optimization techniques
- Unique queue-based rendering management allowing full control over the order of the

rendering process
- Automatic management of object transparency
- Sophisticated and extensible resource management and loading system with included support for file system, ZIP and PK3 archive types

Despite the long list and the extravagant words, the library is fairly easy to use. The following section explains in more details the module architecture.

3.3 Plugin design

As stated in Section 2.2.3, the addition of a new renderer is fairly easy inside Argos. The interface to implement is CRender (see Figure 3.1) which offers few generic methods to implement such as Init() and Destroy(), for module configuration and instantiation, as well as the method Draw() that, as the name suggests, renders the status of the Swarmanoid Space. The interface of a renderer is designed to give the user complete freedom in the choice of the library to use to implement the classes. A renderer can be registered with the macro REGISTER RENDER(class name, label). Similarly to the case of the physics engine, class name is the name of the created renderer class and label is a textual identifier to be used in the XML[3] experiment configuration file.

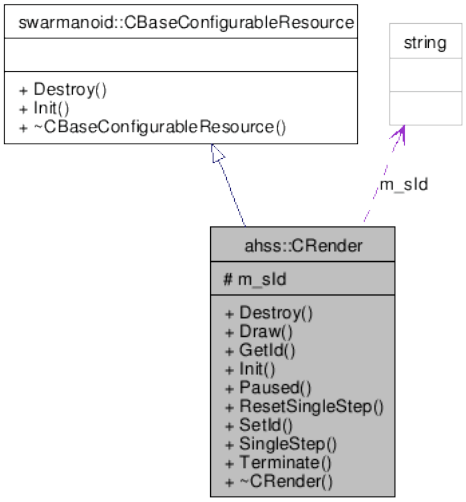


Figure 3.1: The Argos render interface.

[3]The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. `http://xml.coverpages.org/iso15022XML.html`

3.3.1 Code and data structure

The module code is structured around the COgreRender Class, the base class of our rendering module. A more detailed Call Graph can be seen on the Figures 1a and 1.

Module files structure. The small rectangular coloured boxes represent the proportional size of each folder compared to the whole module.

Four folders compose the hierarchy file tree. In *data* we find all needed files for an optimal loading (for more info see (Junker, 2006)). Our module, instead of rendering at each timesteps from scratch, reuses 3D models already stored in the graphical pipeline. This helps us improve the overall speed of rendering compared to a classical draw and flush method.

The *entities* folder has all relevant classes for our home made software reflexion mechanism. It has been implemented using different design patterns such as an Abstract Factory, an Observer and a Composite. The abstract factory allows 3D models to be specifically instantiated, using meshes or simple renderable blocs. The observer notifies each instance of a robot to update itself, while the composite pattern allows us to reuse class behaviours at different points.

Another folder is the *GUI*. All related classes help us increasing the interaction between the user and the simulator.

The last folder is used for our custom 3D mesh automatic loading between the module and a compatible 3D modeller tool.

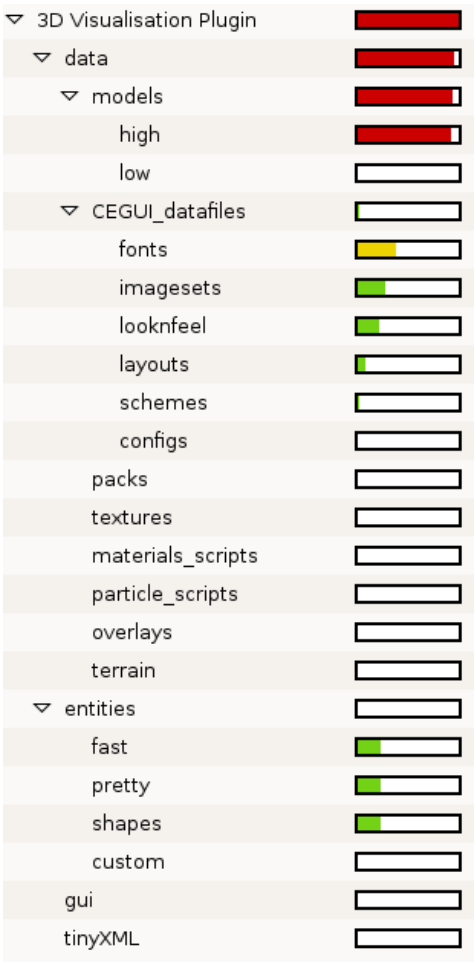
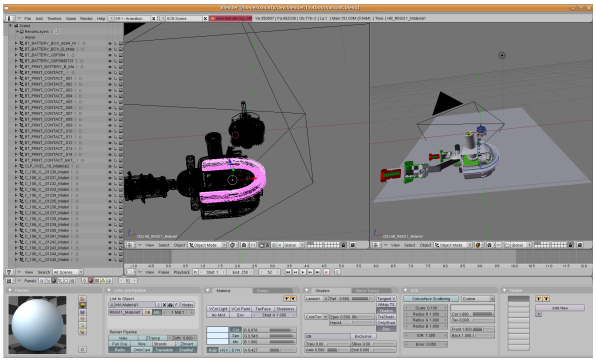


Figure 3.2: Module file structure. The small rectangular coloured boxes represent the proportional size of each folder compared to the whole module.

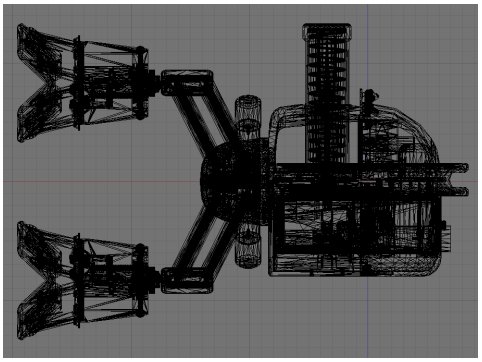
3.4 Swarmanoid entities

Due to the fact that we want to see entities as in real life, we need to have accurate models of the robots. Producing accurate models in 3D of such complex objects as robots, with many mechanical parts is not an easy thing. The practicalities of Swarmanoid project, make it absolutely necessary to use some CAD[4] tool for the design of the mechanical parts, as they are to be produced in a large amount. The sweet deal about this part, is that we can reuse the models that are done with the CAD tools inside our simulator as visual models. The bad side, is that most CAD tools use a proprietary file format for storing their models. As a result, conversion from one format to another needs to be done, and this can vary on the software used. To deal with the numerous different file formats, COLLADA (Arnaud and Barnes, 2006), a royalty free standard 3D asset exchange format, has been created recently. Using this, we where able to convert the CAD models into usable meshes for our module in three steps.

First step needs us to convert the CAD file to a usable 3D mesh. The bad part, is that much informations are lost during this process, and some gets added. Like extra vertices to recreate curves using polygons. After cleaning them, we obtain models such as the one in Figure 3.3.



(a) Building the hand-bot model in a 3D modeller.



(b) Hand-bot wire frame model.

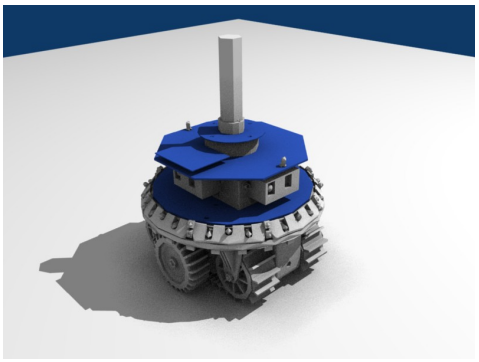
Figure 3.3: The building process of the hand-bot mesh.

Not all the partners of the Swarmanoid project use a CAD tool. Thus, some models had to be done by hand to have a relative nice looking model of the concerned robot.

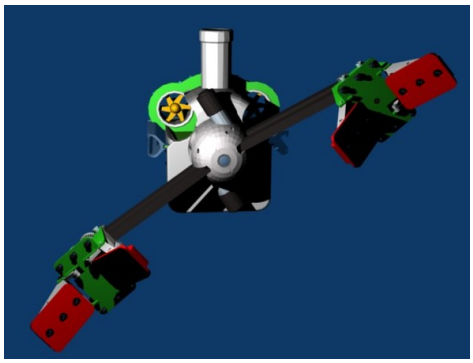
The next step was to incorporate them inside the module. For this purpose, an extra XML parser (see Appendix 2 for an example XML and the 1 for the DTD file) was developed to allow the complex chaining of meshes to be loaded at once. Indeed, as the models evolve a lot, and are still in prototyping, we do not want to hardcore in the library the measures for a forward kinematics positioning of the different subparts that compose a robot.

[4]Computer-aided design

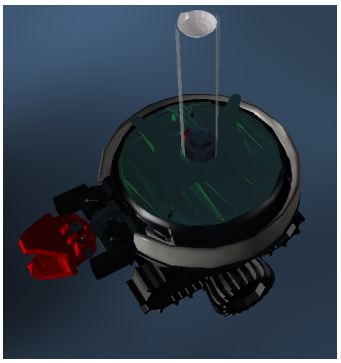
(a) The eye-bot.



(b) The foot-bot.



(c) The hand-bot.

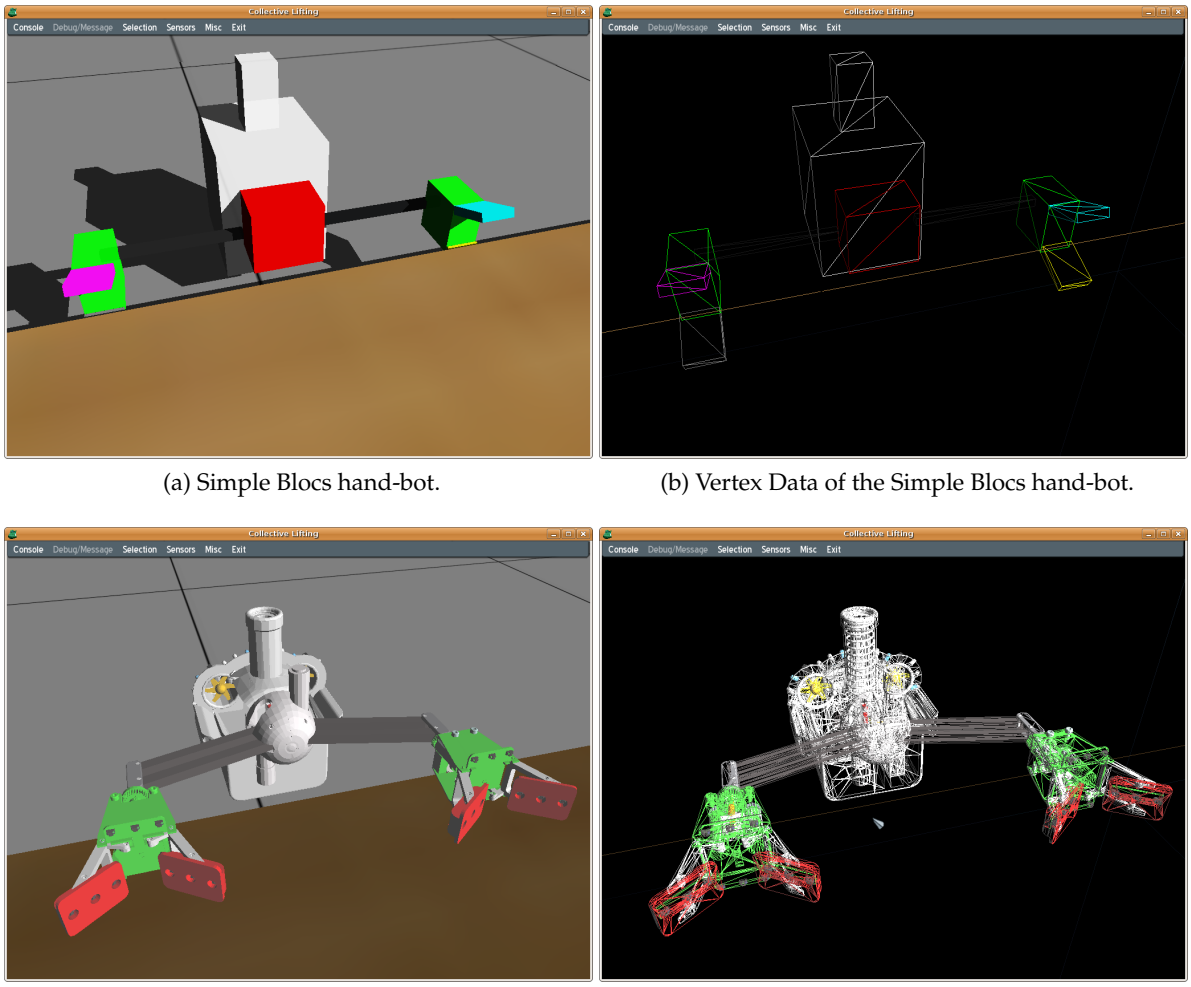


(d) The S-Bot, recreated as it was the first available CAD model.

Figure 3.4: Results of the 3D CAD model conversion.

3.4.1 3D skills

Not everybody is able to model meshes using 3D modellers. It demands graphical skills and time. To overcome the problem, an alternative method to create 3D Object is available inside the module. Basic shapes (circles, cubes, spheres, etc.) are accessible to the end user. These basic shapes can then be used by the end user to create complex 3D objects using a tree hierarchy. The end result of the 3D mesh based model and a basic shape model for the hand-bot can be seen on Figure 3.5.



(a) Simple Blocs hand-bot.

(b) Vertex Data of the Simple Blocs hand-bot.

(c) High Polygon hand-bot.

(d) Vertex Data of the High Polygon hand-bot.

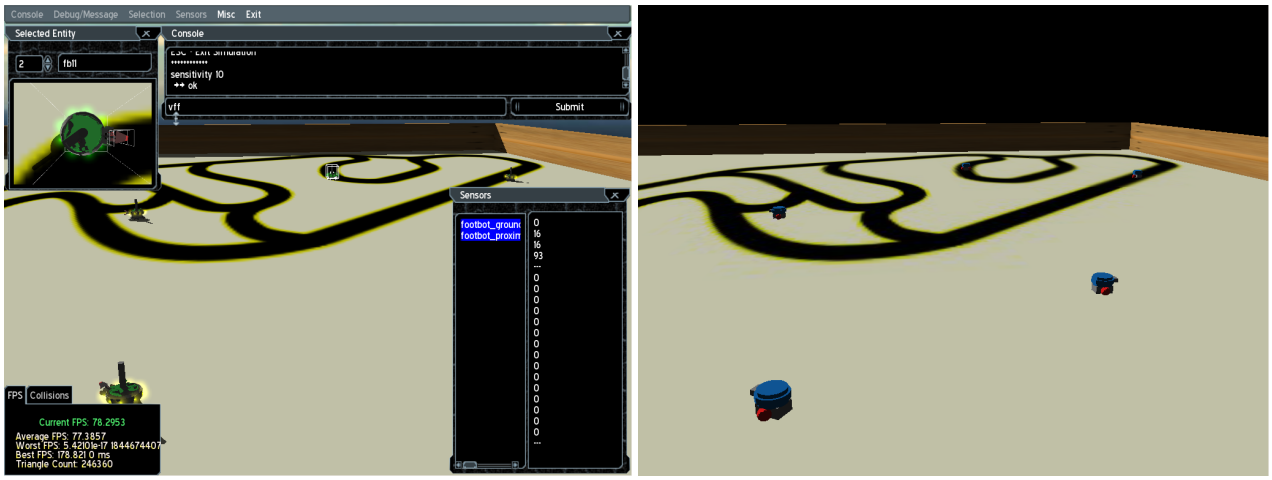
Figure 3.5: Real-time hand-bot models rendered in the simulator.

3.5 GUI

To allow direct interaction between the user and the engine, a graphical user interface (GUI) was need. Indeed, showing robot status, debugging the controllers, moving robots in real time, all of these features require a user interface. For this purpose we chose the CEGUI library. LibCEGUI is a free library providing windowing and widgets for graphics APIs where such functionality is not natively available, such as it is in our case, as ogre is only a rendering engine. The CEGUI library is object orientated, written in C++, thus easy to integrate into our software platform. We decided to embed the GUI inside the rendering process. This was the best choice so to have a working GUI on every operating system. Another choice could have been to embed the whole Argos simulator in a QT[5] application,

[5]see http://trolltech.com/products/qt/

which is also portable on many platforms, but the event processing was not as easy and powerful as the one provided by the CEGUI library.



(a) Ogre module with GUI.

(b) Ogre module without GUI.

Figure 3.6: Module GUI modes.

To speed things up, we did not make the GUI mandatory, as most users only want to see a fast rendering if they have low-end computers. However, by default, the GUI is activated. Figure 3.6 displays two identical Swarmanoid spaces. One is with GUI and uses high polygon models 3.6a, the other is without GUI and uses simple render blocs 3.6b.

The GUI is made of windows. Each window in the GUI is a subclass of COgreGuiWindow (Figure 3.7). This makes it easy to extend the GUI and add new featured windows, which automatically get registered inside a menu. The menu, allows bringing or closing some particular view. Each view adds a new feature that does not automatically rely on other views to work. Almost all features embedded in a view are accessible outside the GUI, making the features available to the users even if the GUI is not instantiated. If the GUI is not instantiated, a minimal console (see Figure 3.8) is still available and can provide the same functionalities as the GUI, but of course, it is not as easy and intuitive. It also requires the

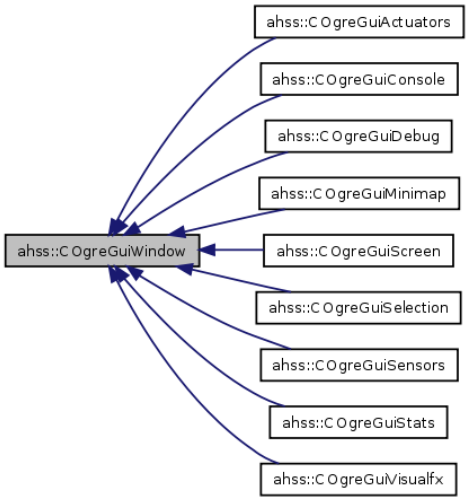
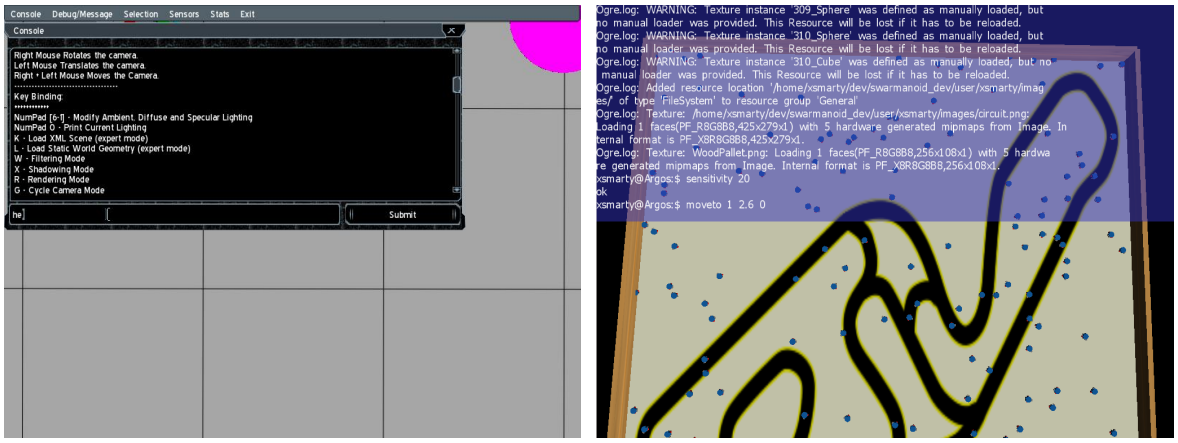


Figure 3.7: GUI Modules.

user to know the correct syntax of each method to call.



(a) GUI console window. (b) Standalone console.

Figure 3.8: Module console.

3.6 Features

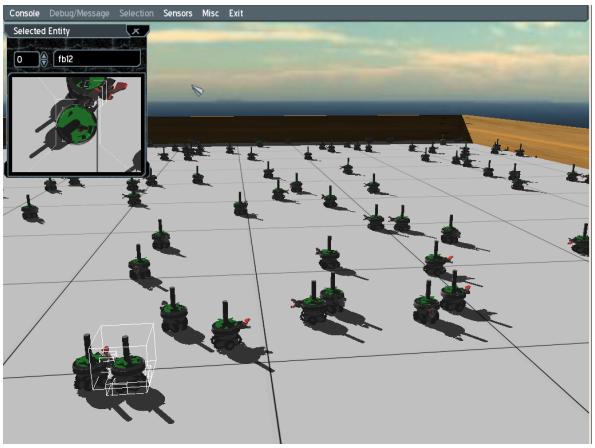
The module offers very useful features for controller development and experiment verifications. The first feature very useful in robotics is the state visualisation of the robot. Such state might be seen using different tools. In Figure 3.9 we display the different possible tools that are integrated to help the development of controllers.

It is frequent that during an experiment, robots do not behave the way we want. They get stuck due to noisy data, or they simply do not handle some special case. Most of the time, the researcher wants to discard, or move this robot to simple test more rapidly a special case. To do so, a pick and move feature was added to the module. As seen in Figure 3.10, the user is able to move a robot easily. This is not a trivial feature, suffice to say that software tracking of the mouse with the frustrum view is the hardest part of the work.

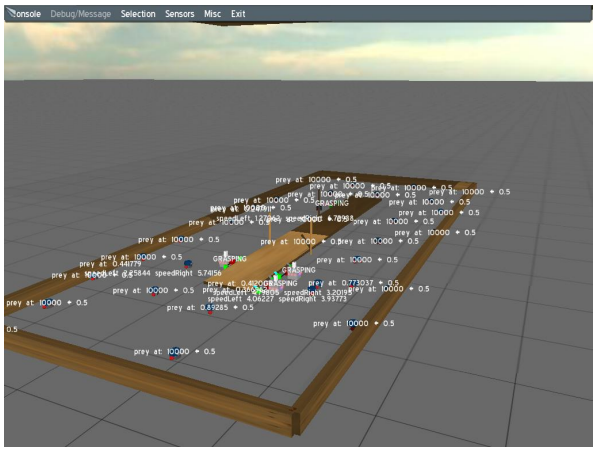
Another feature is the automatic texture management of entities (see Figure 3.11). This allows the user to add a texture to an entity. But this texture, is not just a fancy visual, it actually is part of the Swarmanoid Space and therefore can be used by sensors to read data from the world, such as ground sensors for the foot-bots, allowing them to follow a circuit such as in Figure 3.11d.

As sometimes controllers, and experiments, need to be displayed to the public, movie modes were added. The first mode allows the user an automatic control of the camera directly from the experiment XML configuration file. The user specifies key positions in time, and the module automatically renders a nice movie using those positions as interpolation point of a spline (see Figure 3.12).

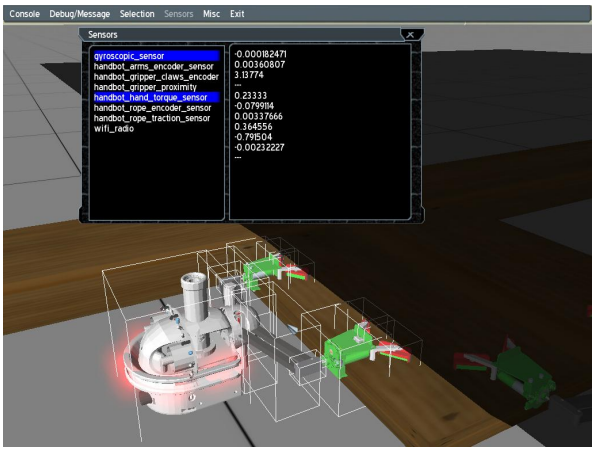
(a) Drawing extra information on top of the entities, on user demand.



(b) Precise entity selection, useful when dealing with a very large number of robots.



(c) Controller state display of all robots.



(d) Sensory data of a selected robot.

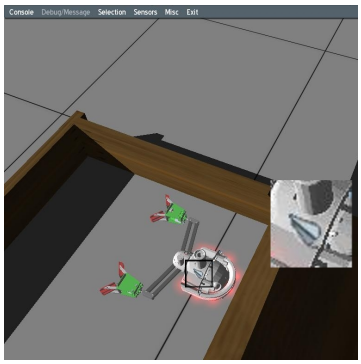
Figure 3.9: View of robots states.

The second mode, makes use of available extra graphical card power, and allows the user to add special effects (see Figures 3.13) that his current hardware can handle. These effects were obtained using GLSL[6] and CG[7] scripts. Apart from the eye candy, this could be a useful tool when using infrared cameras (see Figure 3.13f), as it could simulate the kind of image that such a camera produces for later image analysis in a controller.

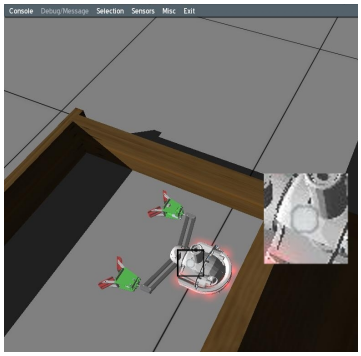
The last useful mode, is the inclusion of any type of scenes using our importer (see Section 3.4). This can lead to nice visual results as the ones show in Figure 3.14. However, this is purely aesthetical, as no entity mapping gets done as we have no physical information of what we include.

[6]The OpenGL Shading Language (GLSL) is part of the core OpenGL 2.1 specification. See http://www.opengl.org/documentation/glsl/

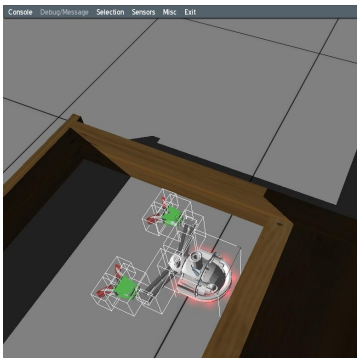
[7]C for Graphics (Cg) is a high-level shading language developed by Nvidia in close collaboration with Microsoft for programming vertex and pixel shaders. It is very similar to Microsoft's HLSL.



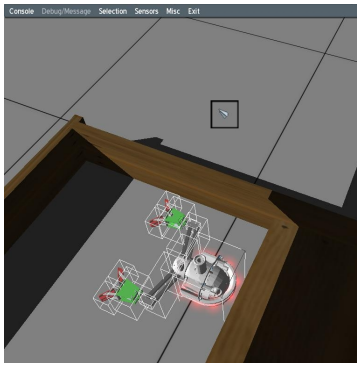
(a) We select an entity.



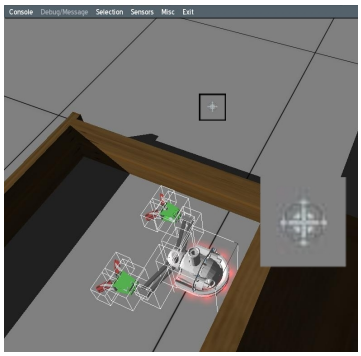
(b) By using control, the cursor changes and allows selection of the entity.



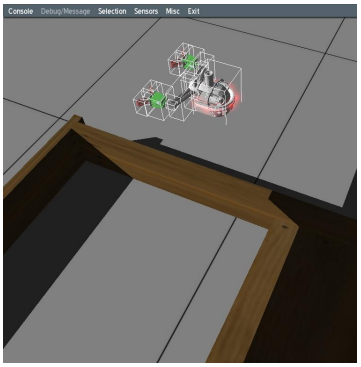
(c) A selected hand-bot.



(d) Choosing a spot to move the hand-bot.

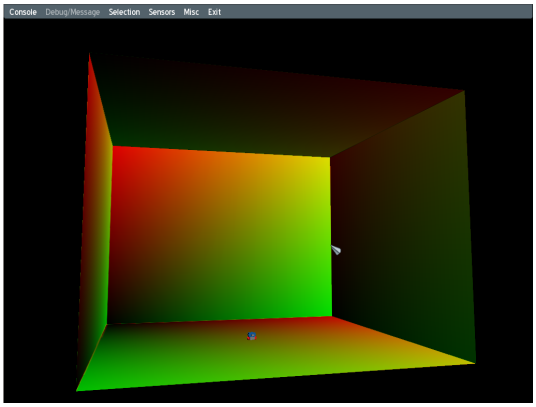


(e) By using alt control, and after a checking process of the feasibility, the cursor changes to an accepted position.



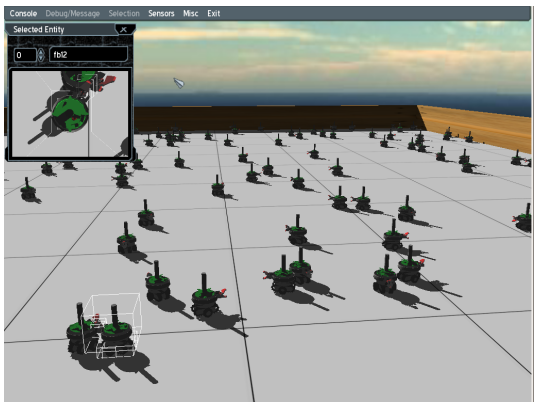
(f) A moved hand-bot.

Figure 3.10: Moving a hand-bot while an experiment runs.



(a) Plane entities placed as a cube, with textures on them.

(b) A texture mapped onto a prey entity.



(c) An experiment with no texture.

(d) An experiment with a sensors readable texture.

Figure 3.11: Texture management of entities.

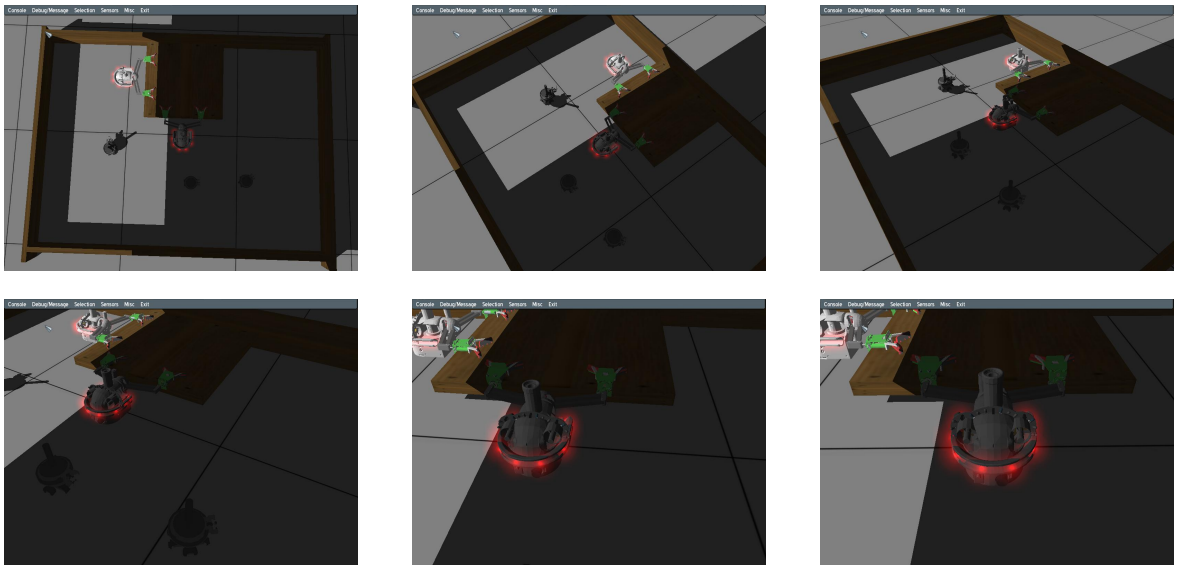
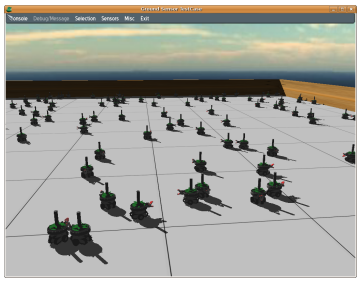
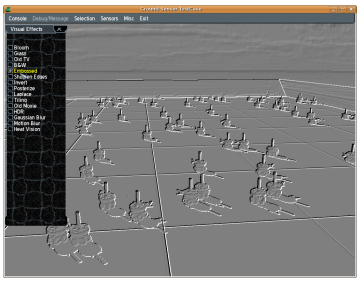


Figure 3.12: Camera travelling.

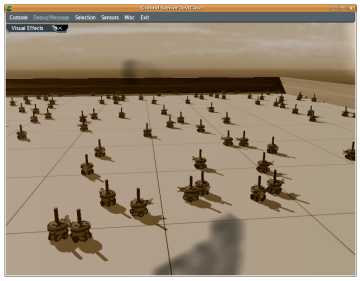


(a) Normal frame.

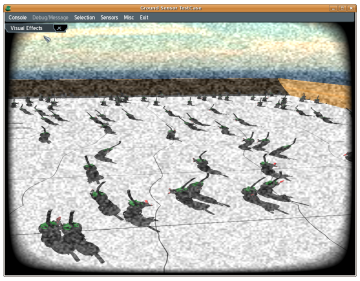


(b) Laplace rendering.

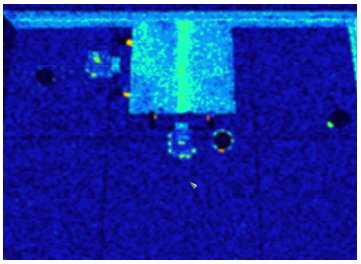
(c) Colour inversion.



(d) Old movie rendering.



(e) Old TV distortion.



(f) Eye-bot simulated infrared camera.

Figure 3.13: Module special effects available.

Figure 3.14: Visual elements added to the simulation.

The next chapter deals with another important module, the 3DPhysics module.

Chapter 4

Physics

Chapter 3 pointed out the reasons why a new and more flexible 3D redering was required for the development of robot controllers in the Swarmanoid project.

The topic of this chapter is the 3D physics module. After a first overall view, the discussion continues detailing the internals of this module.

In more general terms, the component in a simulation program that computes how physical objects should 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. When using a physics engine, an object is explicitly modelled in terms of attributes such as mass, velocity, friction forces, joints, 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 are a number of different ways to represent and implement motion equations, friction forces, collision detection, and equation integration, resulting in a number of different types of physics engines.

At the lower-end in terms of realism, there are physics engines that are mostly kinematics, that is, only first-order dynamics (i.e., velocity-driven) is considered, objects are abstracted by their center of mass, collisions are purely elastic, and friction forces are not taken into account. At the higher end there are physics engines based on accurate descriptions of the objects, of their mechanical constraints and mass characteristics, of the internal and external forces acting on them, and on the solution of the resulting equations using very

sophisticated numerical techniques.

As a matter of fact, even in the higher end class, most of the general implementations of physics engines are restricted to the simulation of *rigid bodies*.

4.1 The state of art

The comparative study of Seugling and Rölin (2006) has taken into account a number of features and performance indices of eight popular physics engines free for non-commercial use (see Table 4.1), shows significant differences among the considered engines.

Table 4.1: Some of the most popular physics engines with their web address. The first eight from the top are free for non-commercial use. Vortex and Havok are commercial products, while PhysicsAndMathLibrary is a non-commercial library for physics-based games.

Open Dynamics Engine	`http://www.ode.org`
AGEIA NovodeX	`http://www.ageia.com`
OpenTissue	`http://www.opentissue.org`
Newton Game Dynamics	`http://newtondynamics.com`
Tokamak	`http://www.tokamakphysics.com`
Dynamechs	`http://dynamechs.sourceforge.net`
True Axis	`http://www.trueaxis.com`
Bullet	`http://www.continuousphysics.com/Bullet`
CM-labs Vortex	`http://www.cm-labs.com`
Havok	`http://www.havok.com`
PhysicsAndMathLibrary	`http://game-physics-engine.info`

The study has been organized in two steps. First, the different engines have been scored according to three basic metrics: (i) supported features, such as collision primitives and inclusion of deformable objects, (ii) availability and characteristics of documentation and examples, (iii) usability, in terms of possibility to integrate the software in other systems, organization of the software, and support for multiple platforms. Second, the three engines overall scoring the best according to these three general evaluation metrics, have been tested on the quality of their response to specific physics issues such as: static and dynamic friction, gyroscopic forces, collision and bouncing, stability of physical constraints, presence of multiple joints, and stability of complex physical contacts.

According to the final evaluation of the authors of the study, *Novodex* (which is in reality 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.

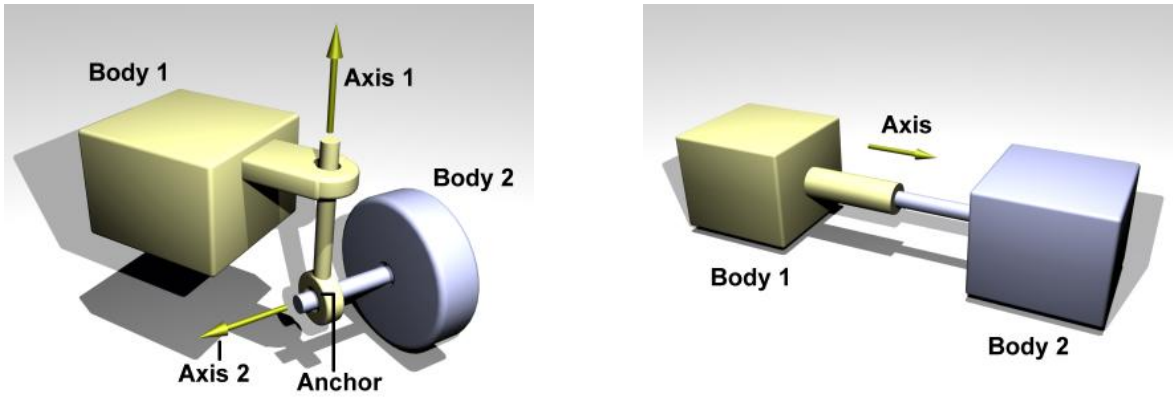
Clearly, these mentioned engines all belong to the class of engines providing high accuracy and full 3D dynamics simulation. It is worth to mention that AGEIA has been also the first to produce a physics processor, the PhysX chip, which has a PPU (Physics Processor Unit) specifically designed for physics simulation.

The physics engine we chose is ODE because it is at a mature stage, it is free and has a performance comparable to the best commercial packages.

4.2 3D dynamics plugin design

The ODE library supports rigid body dynamics with an arbitrary mass distribution. Bodies are connected to each other via joints. A group of bodies is a group where bodies can not be pulled apart: every body is connected somehow to every other body in the group. Each group in the world is treated separately when the simulation step is taken. This is a crucial point for optimizing simulations. A very stable first order integrator is used, meaning that the calculation error does not cause the system to have non-physical behaviours. However it is not particularly accurate, at least not enough for quantitative simulations, unless the simulation step size is very small[1].

Many joint types are supported such as the ball-and-socket, the hinge, the slider, the hinge-2 (car wheel), the fixed joint, the angular motor, the universal and the contact joint. When two objects collide each other, a contact joint is created, which makes sure that the two geometries do not penetrate each other. Very large groups can be created in this way.



(a) Hinge-2 joint.

(b) Slider joint.

Figure 4.1: Example of available joints in ODE.

The available collision primitives are: sphere, box, capped cylinder, plane, ray, triangular mesh. We notice the absence of a normal cylinder and a heightmapinside these. Table 4.2 show information about inter shapes collision detection. To speed up collision detection, it

[1] in the order of 2 ms

is important to reduce the number of geometries as much as possible, that is why a robot is modelled using the least possible primitives.

	Ray	Plane	Sphere	Box	Capsule	Cylinder	Trimesh	Convex	Heightfield
Ray	NO	YES	YES	YES	YES	YES	YES	YES	YES
Plane		NO	YES	YES	YES	YES	YES	YES	NO
Sphere			YES	YES	YES	YES	YES	YES	YES
Box				YES	YES	YES	YES	NO	YES
Capsule					YES	NO	YES	NO	YES
Cylinder						NO	YES	NO	YES
Trimesh							YES	NO	YES
Convex								YES	YES
Heightfield									NO

Table 4.2: Collisions detection among each primitive inside ODE.

The friction model used is an approximation of Coulomb friction model. Two time stepping methods can be used: worldstep and quickstep. Worldstep, which solves the system of constraints by inverting a matrix, takes a lot of time and memory, however the results are very accurate. The second method is called quickstep, it is an iterative constraint solver method. The number of iterations can be chosen and the accuracy to speed ratio can be controlled that way. Quickstep is only really efficient if all the connected bodies have the same mass density close to 1 kg/m^3. We mainly use worldstep, however the user can still choose what time stepping method he will use.

The next chapter explains in a more detailed manner, how each sensor and actuator was modelled.

Chapter 5

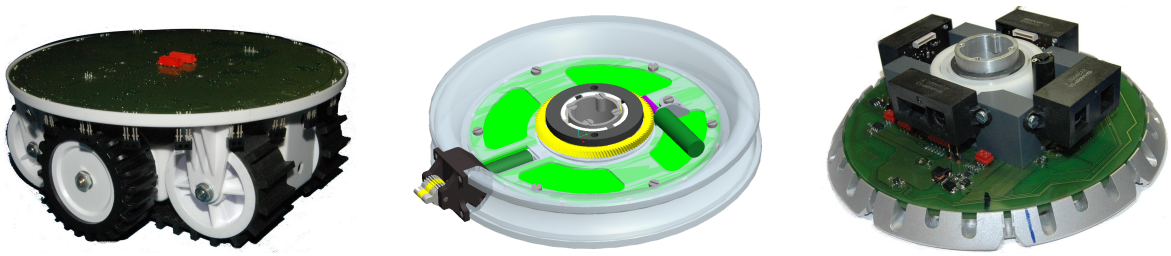
Simulated sensors and actuators

The topic of this chapter is the implementation of the real robot sensors and actuators. We will explain into details the relevant features used in the experiment of Chapter 6. Due to the fact that all the robots presented here are still into prototyping, many values are still not available. This has led us to use parametric values in all simulated devices. Proper validation will need to be done when real data is available.

As an architectural feature, we chose to add noise on every actuator and sensor. The way noise is added to the readings is configurable. Most sensors use by default Poison or Normal noise distributions. However, the user can choose to alter this through the XML configuration.

5.1 Foot-bot sensors and actuators

The foot-bot is composed of 3 separated levels: the base, the docking module and the top module.



(a) Base module. (b) Docking module. (c) Top module.

Figure 5.1: Modules of a foot-bot.

The base module provides energy and basic mobility to support the higher features of the foot-bot. The docking module allows self-assembly between foot-bot and other capable robots. The top module supports most of the sensors available on the robot.

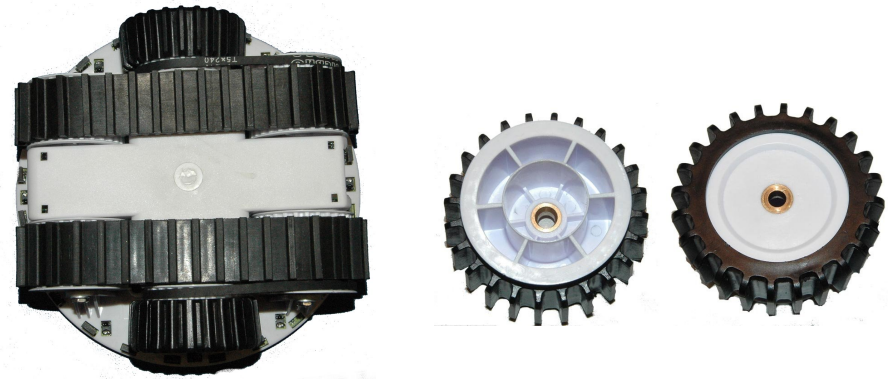


Figure 5.2: The treel driving system (left). Details of the wheel (right).

5.1.1 Mobility

Treels actuator

The Treels[1] provide mobility to the foot-bot (Figure 5.2). They consist of two 2 W motors, each associated with a rubber track and a wheel. Motors are driven by dedicated boards and maximum speed of the foot-bot is 30 cm/s.

They are implemented using a 3D physical model taken from the previous Swarm-bot project. Friction of the rubber on different surfaces is however not yet known. The actuator that implements the interface to control them is linked to the ODE physical module.

Odometry sensor

The odometry for the treels has been implemented as a wheel rotation encoder. It is generic, meaning that we do not need to link it against a physical engine as no physical property is altered in the process of reading the values. They are computed using the robot position, rather than reading the actuator.

5.1.2 Proximity sensors

The base of the foot-bot includes infrared sensors to act as bumpers and ground detectors (Figure 5.3). These sensors have a range of some centimeter and are distributed around the robot: 24 are directed outside and 8 are directed to the ground. In addition, 4 contact ground sensors are placed under the lowest part of the robot.

Ground sensors have been used in conjunction with texture loading, to help us simulate different ground colours. The proxy sensors and the ground sensors are done generically, they do not need physics as no physical property is altered. As a remark, sensors obtained are not perfect, as in the simulator they sense only following a direct line, while in real life,

[1] Treels is a contraction of TRacks and whEELS

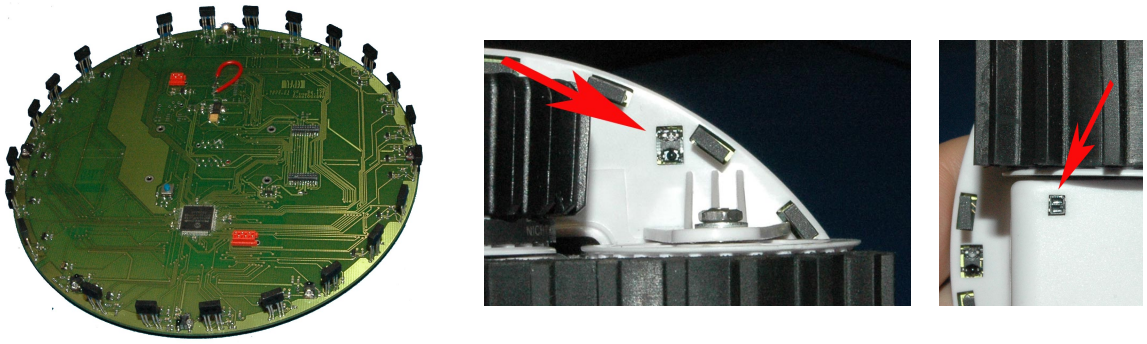


Figure 5.3: The current prototype of the foot-bot sensors board (left). Ground proximity sensor (center). Contact ground sensor (right).

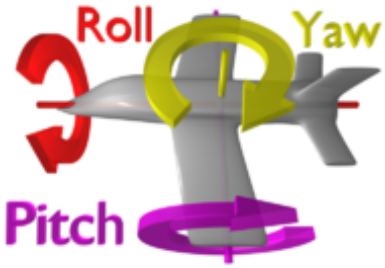


Figure 5.4: Gyroscopic sensors help keeping track of the current rate of acceleration on each axis.

the sensed area is cone shaped. Increasing from directional to cone shaped sensing would demand lots of computation depending on the wanted precision.

5.1.3 Gyroscopic sensor

To compensate for the bad odometry the treel subsystem provides (because of the tracks), the foot-bot includes an IMU[2]. This IMU will provide 3D odometry (Figure 5.4).

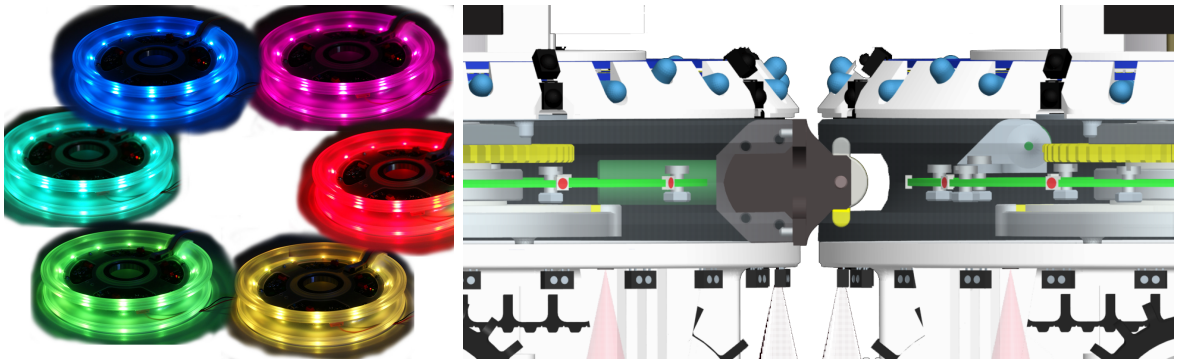
5.1.4 Docking module

The docking module allows foot-bots to connect to each others. It is a moulded plastic part that integrates a traction sensor and 12 RGB LED's (Figure 5.5b). It is also able to rotate with respect to the base and the top modules.

LEDs actuator

The LED's[3] state is written directly in the Swarmanoid space entity. It is generic.

[2] An inertial measurement unit.
[3] A light-emitting diode (LED).



(a) The colour ring of foot-bot.

(b) The foot-bot docking module prototype parts.

Figure 5.5: Docking module of a foot-bot.

Gripper actuator

We used a sticky gripper for the implementation of this module. Since it requires interaction among physical bodies, it is linked to the physical engine.

5.1.5 Rotating distance sensor

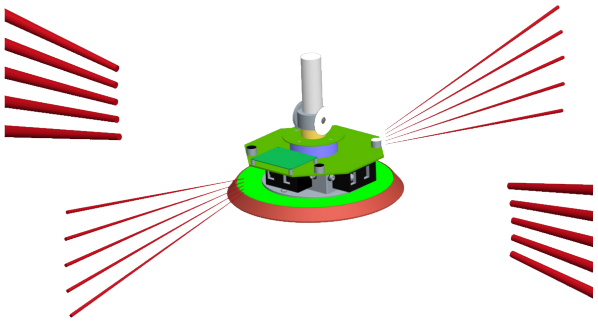
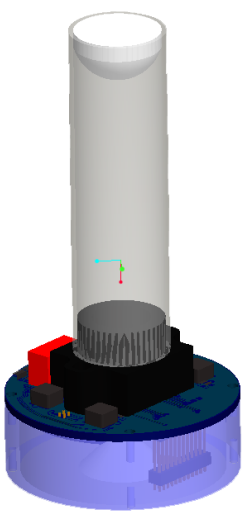


Figure 5.6: The foot-bot rotating distance sensor.

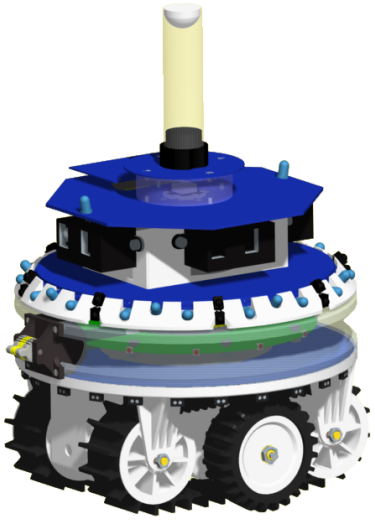
The rotating sensor scans the environment. Too few information was available, so we implemented them, in a generic way, as if they where long range 1D IR cameras.

5.1.6 Omnidirectional camera sensor

The omnidirectional camera was implemented using a generic sensor that detects the nearby coloured entities. LEDs are easily detected in real life using a colour segmentation algorithm. The distance and bearing of those LEDs can then be extracted. We implemented the same behaviour.



(a) The foot-bot omnidirectional camera.



(b) The foot-bot.

Figure 5.7: The foot-bot camera sensor.

5.1.7 Communication module

A communication module is available through WIFI, IR and RFID. Proper description is not yet available as prototyping continues. However, a generic WIFI model was implemented. It can be used with maximum range parameterisation, thus allowing us to act as if we had local or global communication.

5.2 Hand-bot sensors and actuators

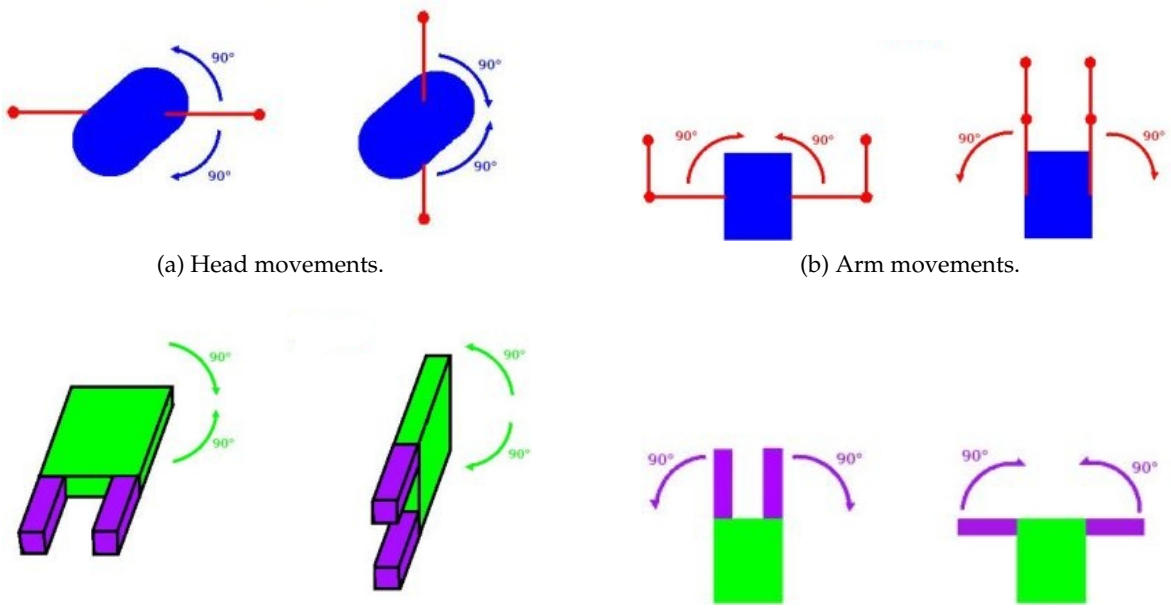
The hand-bot is composed of roughly 5 parts: the hull, the rope launcher, the head, the arms(2) and grippers(2). The hand-bot is able to move its parts following Figure 5.8.

5.2.1 The hull

The hull (Figure 5.9) contains, just as the foot-bot, proximity sensors, gyroscopic sensors, a docking module, LED's and a communication module. Each of them have been explained in Section 5.1.

5.2.2 Rope launcher

The rope launcher is made of 4 motors: one for recharging the spring and unlocking the launch, one for the fast rewind of the cable when the magnet is detached, one for moving



(a) Head movements.

(b) Arm movements.

(c) Gripper movements.

(d) Gripper Claw movements.

Figure 5.8: The possible movements of the hand-bot.

the hand-bot up and down the rope, and one servo-motor for connecting the previous motor to the rope driving system (Figure 5.10).

Rope actuator

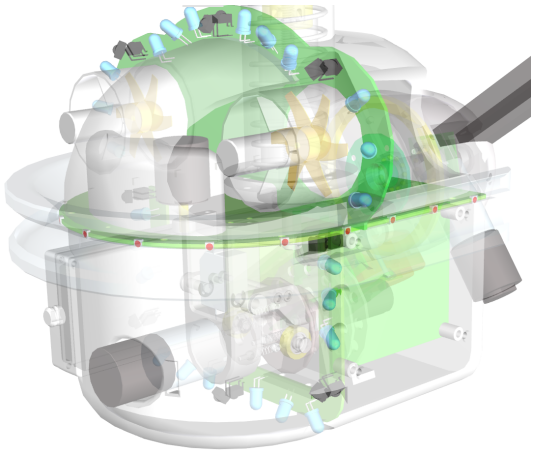
The rope was made using a slider joint, to allow the translation on the axis, and a ball joint as the anchors rotation points (Figure 5.10c).

Rope traction sensors

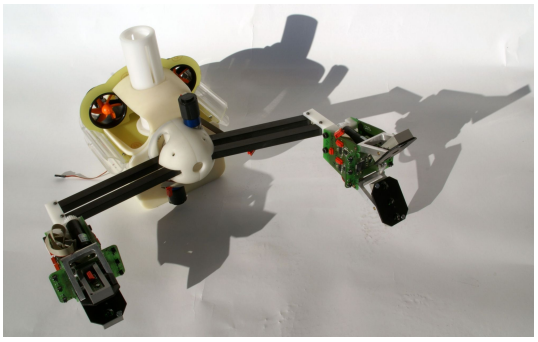
Dynamical traction sensors where added to the model, to simulate the traction on the rope engines. This is very handy when we want to know if the hand-bot did attach correctly, and to know if he is lifting extra weight when going up.

5.2.3 Arms and head

The head of the hand-bot is the attach and rotation point for the arms. At the top of the head, a camera with a fish-eye lens, fixed with respect to the body, is place. Two powerful motors are driving each arm independently. There should be able to push the robot away from the shelters when it is attached at 1 m from the ceiling (Table 5.1). A third motor is responsible for rotating the arms with respect to the body. It is designed to let the robot turn the head when it is grasping an object (Table 5.2).



(a) The hull of the body of the hand-bot.



(b) The current hand-bot prototype.

Figure 5.9: View of a hand-bot.

The camera, the motor actuators and encoders, where implemented using same techniques as in previous section.

Motor running at 3,0V	Theory	Measured
Nominal torque (with nominal speed)	0.72Nm	
Nominal speed (with nominal torque)	2.53 rad/s	2.29 rad/s
Stall torque		2.27 Nm
No load speed		2.61 rad/s

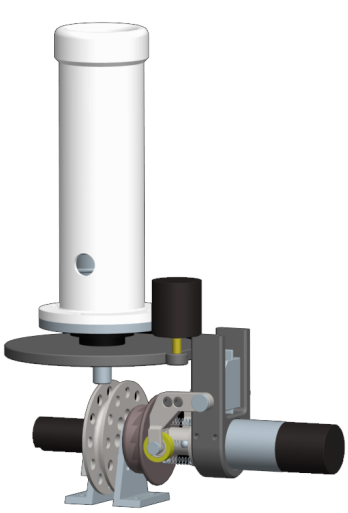
Table 5.1: Characteristic of hand-bot arms.

Motor running at 3,0V	Theory	Measured
Nominal torque (with nominal speed)	0.32 Nm	
Nominal speed (with nominal torque)	1.77 rad/s	1,54 rad/s
Stall torque		1.44 Nm
No load speed		2.85 rad/s

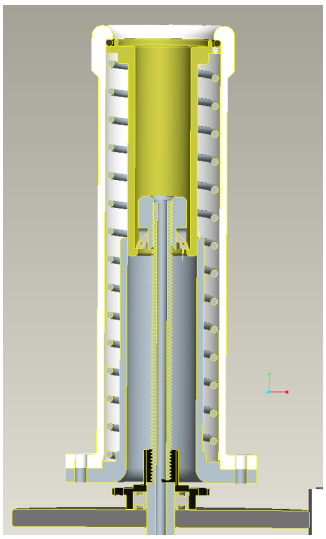
Table 5.2: Characteristic of hand-bot head rotation.

5.2.4 Grippers

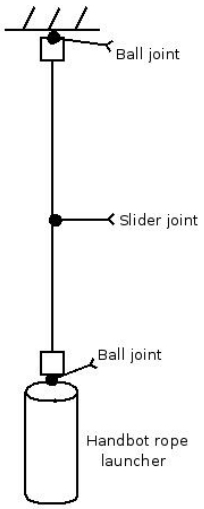
The gripper is placed at the end of the arm. It consists of two motors: one for rotation and one for grasping. The gripper also embeds a low resolution colour camera (VGA) and 12 distances sensors in order to locate and grasp environment and objects. The gripper was designed so that it can support the weight of the robots when the arms are in a vertical position. This implies a high grasping force of 25 N (Table 5.3). The gripper can also rotate with a load of 2 N (weight of a book). See Table 5.4. Additionally, the gripper contains 24



(a) The rope launcher hand-bot prototype.



(b) The rope launcher hand-bot prototype interior.



(c) Modelled rope.

Figure 5.10: The rope launcher of the hand-bot.

proximity sensors for accurate distance sensing.

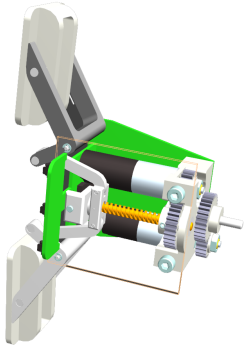
Gripper and gripper claw actuators, torque sensors and encoders where properly modelled generically. The proximity gripper sensors where done in two flavours, a simple 3 proximity sensors, and a full 24 sensors per hand.

Motor running at 3.0V	**Theory**	**Measured**
Nominal force (at finger tip axis, with nominal speed)	25 N	
Nominal speed (at finger tip axis, with nominal torque)	0.013 m/s	Almost 0 m/s
Stall force		24 N
No load speed		0.03 m/s

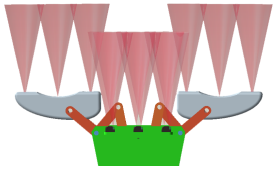
Table 5.3: Characteristic of hand-bot gripper's claw.

Motor running at 2.5V	**Theory**	**Measured**
Nominal torque (with nominal speed)	0.08 Nm	
Nominal speed (with nominal torque)	0.524 rad/s	0.491 rad/s
Stall torque		265 mNm
No load speed		0.604 rad/s

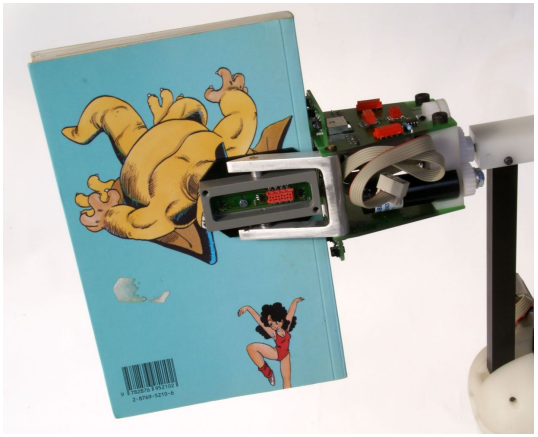
Table 5.4: Characteristic of hand-bot gripper's rotation.



(a) The CAD gripper model of the hand-bot.



(b) The proximity sensors in the gripper of the hand-bot.



(c) The gripper of the hand-bot, grasping a book.

Figure 5.11: The gripper of the hand-bot.

5.3 Eye-bot sensors and actuators

The ceiling attachment mechanism actuator (Figure 3a) , the distance scanner(Figure 3b) and the optical flow sensor(Figure 3c) are not yet implemented. See Appendix .3 for more information.

5.3.1 Propulsion actuator

The eye-bot platform is based on a quad rotor design (Figure 5.12). The propulsion system consists of a brushless outrunner motor and a propeller. Each motor and propeller is capable of supplying a thrust of up to 350 g, thus giving a total thrust capacity of 1.4 Kg.

5.3.2 Vision sensor and laser actuator

The system is capable of panning 360 degrees in the horizontal plane and tilting 90 degrees from vertical to horizontal. There will be a 2.0 mega pixel CMOS colour camera and a red 5 mW laser module mounted on the end of the articulation (Figure 5.13). The camera resolution will be adjustable through software using a windowing method which is part of the i.MX31 capture system, allowing for a digital zoom functionality. The laser will be used as a pointing device to identify targets to the foot-bots while looking down from the ceiling. A red-green-blue (RGB) colour signalling light will be placed directly on the bottom of the eye-bot which can be used as a visual communication device between robots.

The camera and the laser pointer are generically available as actuators.

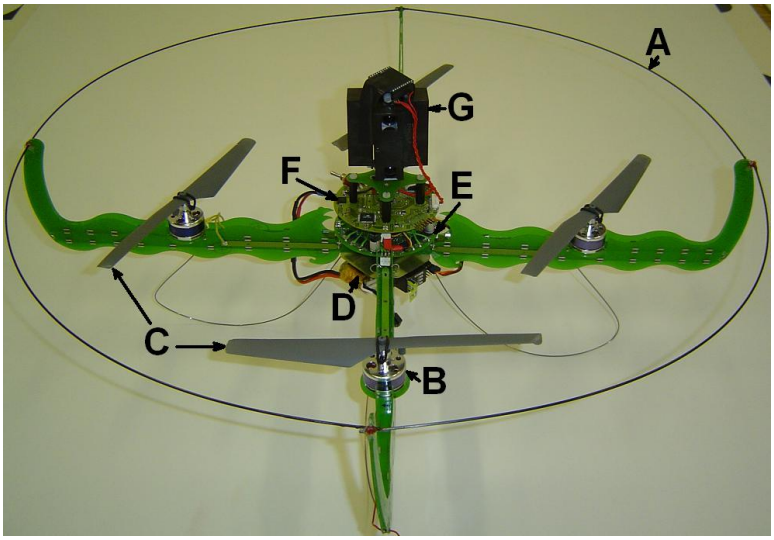


Figure 5.12: Current quad rotor platform: A.) collision protection ring, B.) brushless motor, C.) contra-rotating propellers, D.) LIPO battery, E.) high-speed motor controller, F.) flight computer, G.) infrared distance sensors and ceiling attachment.

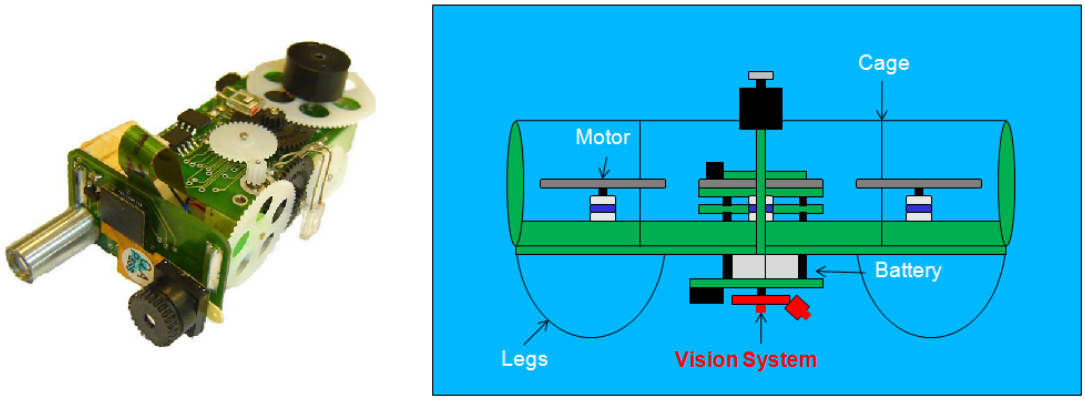


Figure 5.13: Omni-directional vision system: left - prototype, right - mounting location.

Chapter 6

Simulated experiments

The aim of this chapter, is to show a working experiment inside the Argos(see Chapter 2) framework. In Section 6.1 an overview of the problem we want to solve is done. To solve it, collaboration between hand-bots and foot-bots is needed. Two controllers have been developed and we present them in Section 6.2, as well as we present the obtained results. Finally, a methodology for behaviour shaping inside the platform is revealed.

6.1 Scenario

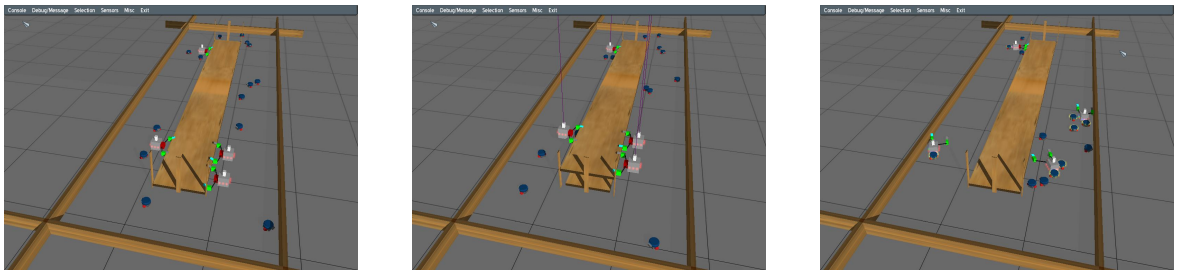
6.1.1 Motivation

One of the final goals of robotic systems is to help us, humans, to ease our lives. For this purpose, and to overcome the complexity of our world, we take the swarm robotics approach. But creating minimal rules that make an autonomous cooperative swarm solve a task is not trivial. The biggest difficulty is to predict what kind of individual rules will produce a desirable effect on the whole swarm. Using hard-coded behaviours works fine when we have knowledge of the environment, however these solutions lack adaptability, a feature that most classical robotic systems fail to acquire. Evolutionary techniques (see (Jong, 2002)), on the other side, tend to be more adaptive. However, blending evolutionary techniques with traditional hand-coded simple behaviours might be even better. Before running evolutionary algorithms that will select the best parameters for the swarm task problem solving, some basic behaviours have to be coded. One can hard-code these. Once some behaviour is found suitable, we use evolutionary techniques to find the best possible match. For this purpose, we chose a scenario where we could test those basic hand-coded behaviours.

6.1.2 Overview

In our scenario (Figure 6.1), a heavy object is on the floor. It needs to be lifted by the hand-bots. The initial hand-bots geometry disables them to move this object as they are in imbal-

anced positions. Imagine 10 persons lifting a piano, but 9 of them are on the same side. To balance the whole we need to move the hand-bots around our heavy object on the floor. This is where the foot-bots come into play. They have to grab the hand-bots and move them until a correct geometry allows the object to be lifted. Eye-bots were discarded at the moment. As our scenario needs homogeneous and heterogeneous collaboration and fits perfectly inside the Swarmanoid project.



(a) Control step 400. (b) Control step 2500. (c) Control step 427700.

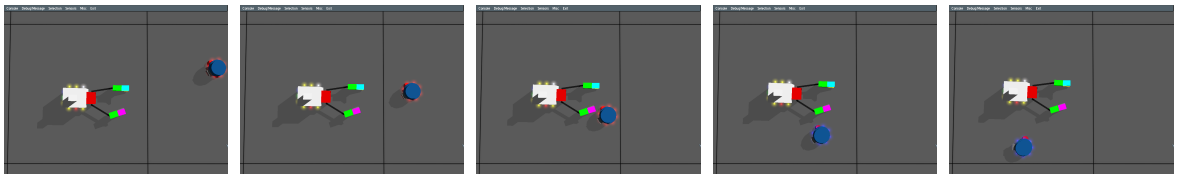
Figure 6.1: Full scenario trial with 4 hand-bots and 14 foot-bots.

6.1.3 Methodology

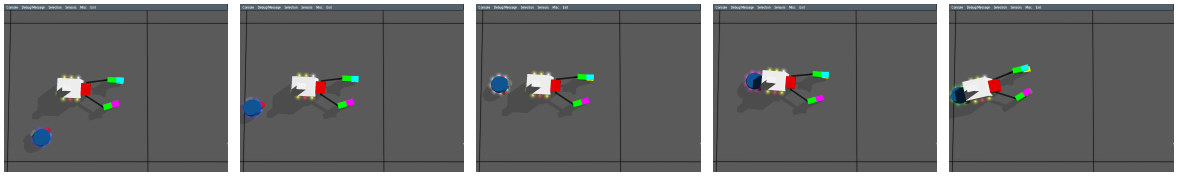
To solve our scenario, we identified a set of subtasks that needed to be achieved. We classified them by complexity of coding, and started to solve them in simulation. First, sensors and actuators were added to the simulator. Later, small subtasks were solved inside small special cases experiments (like gripping a book with the hand-bot, moving around a foot-bot without colliding with walls or grasping at a precise spot (Figure 6.2)). Then, we fused the developed controllers into one heterogeneous experiment, our scenario. From this point, controller development became parallel, as colaboration was expected between the two types of robots. The final step, was the finding of proper parametric values that triggered behaviours, using evolutionary selection and fitness. Next section explains the obtained controllers and results.

6.2 The Push-Pull controller

To solve this scenario, we established symbiotic controllers between foot-bots and hand-bots. We call them the Push-Pull controllers. The name will become evident as the controllers get detailed.



(a) Control step 425. (b) Control step 605. (c) Control step 785. (d) Control step 965. (e) Control step 1145.



(f) Control step 1325. (g) Control step 1505. (h) Control step 1685. (i) Control step 2045. (j) Control step 2195.

Figure 6.2: A foot-bot connection to a hand-bot at a precise location (green-blue slot).

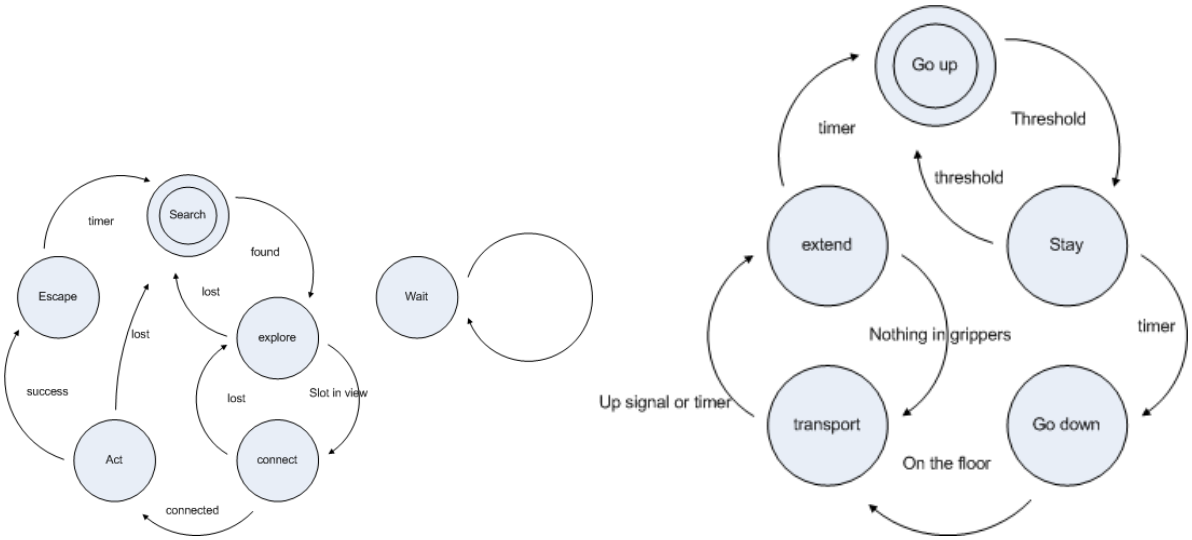
6.2.1 Foot-bot control

The foot-bot controller can be seen on Figure 6.3a. At first a foot-bot searches using its camera for coloured LEDs. If he finds something, he tries to investigate. During the investigation, if some slot is found then he connects to it. A slot is formed by two different coloured LEDs, a blue and another colour. The foot-bot will connect to nearest slot available to him. Depending on the slot type, he then pushes or pulls. If the slot is blue-green, he pulls. If it is blue-violet, he pushes. When he is doing something interesting (connecting, pushing or pulling) he warns nearby foot-bots that he is busy, by lighting all it's LEDs in yellow. This helps others foot-bots not to interfere. However, a small complexity was added to the controller to allow multiple foot-bots to work on the same hand-bot. Indeed, one hand-bot can have multiple slots, and this can create problems. One of them is a foot-bot trying to connect to a slot, while the hand-bot is already being moved. This can require the connecting foot-bot to track its target slot, thus integrating its position over time. A simpler solution to this problem was the use of a MACA[1] (Karn, 1990) variant.

If a foot-bot wants to connect to a hand-bot, it sends a local signal that makes other possible active foot-bots pause their current activity. This gives enough time for the current foot-bot to connect to its target hand-bot. Once the connection is done, all foot-bots are unpaused. Should the connection be severed, or take too much time, foot-bots are automatically unpaused after some time. The time on hold, as well as the local communication radius are parametric.

Foot-bots can also be in a "wait" state. This state places a foot-bot on hold until the counter-order is given. The trigger to this are messages received from the hand-bot when

[1]Multiple Access with Collision Avoidance (MACA) is a slotted media access control protocol used in wireless LAN data transmission to avoid collisions caused by the hidden station problem and to simplify exposed station problem.



(a) Foot-bot FSM. (b) Hand-bot FSM.

Figure 6.3: The Push-Pull finite state machines.

they lift and they go down. This prevents foot-bots from being squashed by hand-bots as they go up or down. Of course, the hand-bot needs to minimise its oscillation when going down and land at the same position. All possible controller parameters (speed wheels, radius state changes, etc.) are parametric.

6.2.2 Hand-bot control

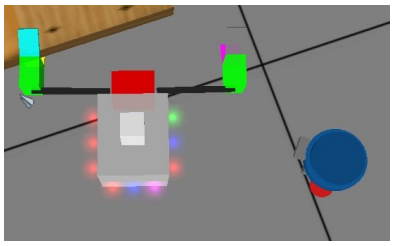
The hand-bot controller can be seen on Figure 6.3b.

At anytime, a hand-bot lights up a push slot (Figure 6.4b) in the back as soon as it does not detect something in the gripper.

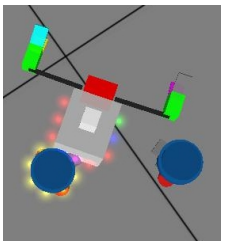
Randomly, a signal is broadcasted by a hand-bot. If a hand-bot receives this signal (from another, or from itself), the hand-bot starts a "try to lift" procedure. It will go up indefinitely, by climbing on its attached rope, unless some threshhold is reached. If too much rope traction (object is too heavy), no rope traction (no ceiling attachment) or gyroscopic threshold is sensed (unbalanced geometry), the hand-bot stops lifting and goes back down. Depending on the threshold type, the hand-bot will open a slot to be moved to a new position (Figure 6.4a). Again,

6.2.3 Results

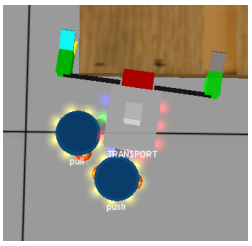
The experiment was conducted with two hand-bots and a varying number of foot-bots. Figure6.5 details the whole experiment. The obtained results can be see on Figure6.6. Wider experiments were done, with different arena sizes and robots geometry (Figure 6.1) but no



(a) Foot-bot connecting to a pull slot.



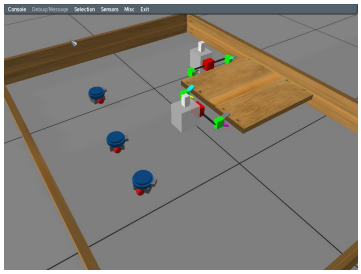
(b) Foot-bot pushing a hand-bot.



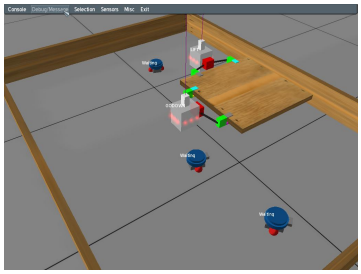
(c) Two foot-bots doing a Push-Pull.

Figure 6.4: Interaction between a hand-bot and two foot-bots.

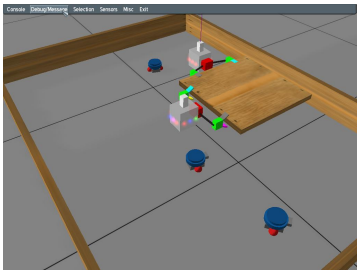
statistical data was extracted for them due to lack of time, and the fact that real robot design will change in coming months.



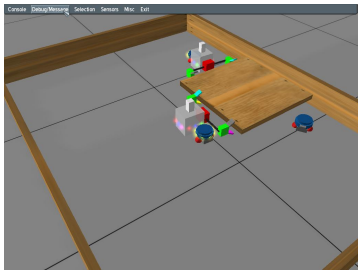
(a) Control step 24.



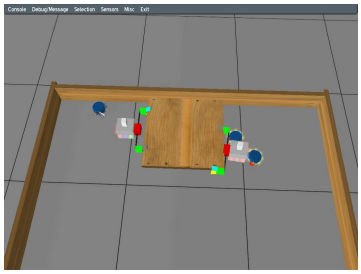
(b) Control step 1368.



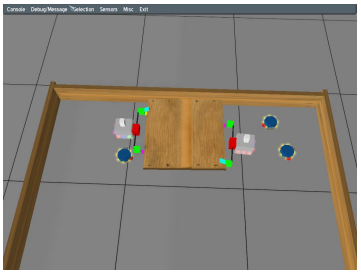
(c) Control step 1464.



(d) Control step 2328.



(e) Control step 11736.



(f) Control step 36144.

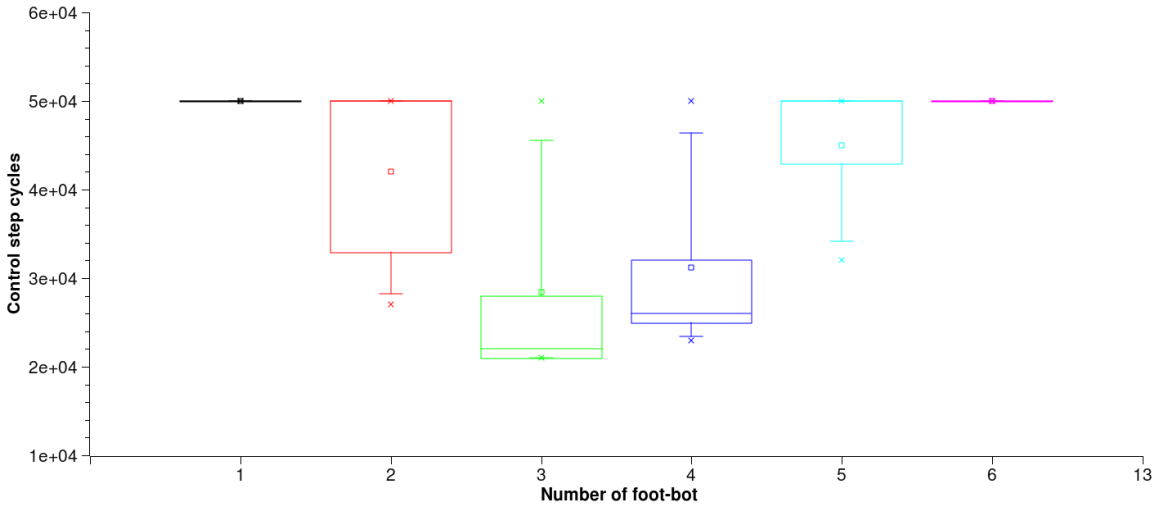
Figure 6.5: Successful experiment with two hand-bots and three foot-bots.

The obtained controllers, as well as the corresponding platform source code, is available at under a versionned SVN [2] server. Our results have been obtained with revision 1680.

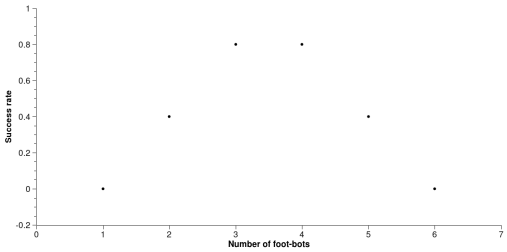
Analysis

The first observation to be made, is that below, or above a certain foot-bot density, the experiment fails. The arena of 5m had an exclusion zone of 2 m^2, leaving 3 m^2 for foot-bot

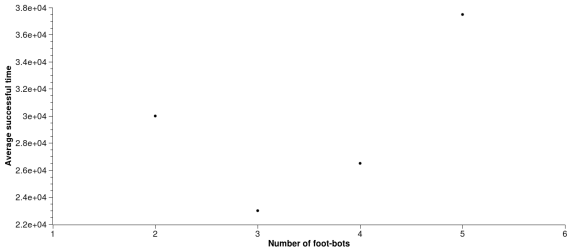
[2] `http://www.swarmanoid.org/~swarmanoid/mediawiki-1.5.2/index.php/Documentation`.



(a) Time to complete the whole experiment. No data above 50000 cycles, as it was our threshold for experiment failure.



(b) Success rate of the experiment.



(c) Average time for successful experiment.

Figure 6.6: Obtained data from a Push-Pull with two hand-bots and different foot-bots numbers.

placement. If the density was above two foot-bots per m^2, too many collisions, and interference made the whole system quite slow, thus achieving easily the time threshold of 50000 cycles. Too low density made foot-bots spend too much time on exploration.

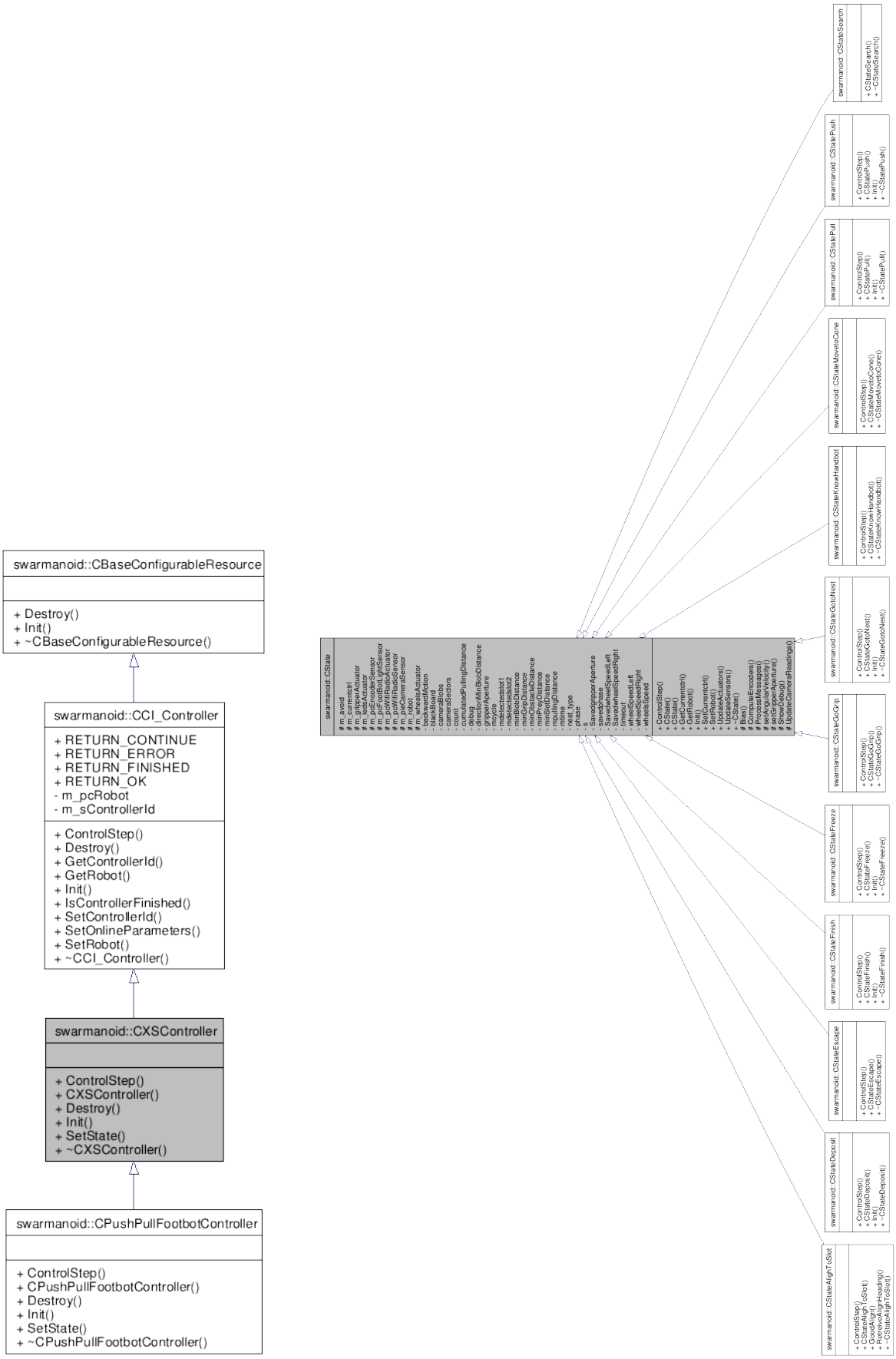
As a second observation, we can say that the success rate is still strongly bond to initial topology. This infers the need for deeper parametric exploration of the controllers to obtain better results.

Finally we can state that the Push-Pull mechanism works.

6.3 Reusable behaviour design

For the purpose of our control, instead of using a flat class with C++ switch-cases, that becomes impossible to work with as a controller expands, we switched to a state pattern controller (Figure 6.7). This helps us encapsulate each behaviour as a separate class that we can

then reuse on any other controller, allowing easy code reuse.



(a) The controller.

(b) The states.

Figure 6.7: An OO state pattern controller.

Chapter 7

Conclusions

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

In this work we have presented a new robotic concept, called a Swarmanoid, made of an innovative distributed robotic system comprising heterogeneous, dynamically connected small autonomous robots. We exposed the simulation framework, and the associated developped tools: 3D visualization and a 3D physical engine. We explained in details each of the robots and their simulated counter part. Also, as a validation, we presented the results obtained in an attempt to control the Swarmanoid. We chose a mixed approach in the methodology of the controller development. Simple behaviours were extracted, and refined using evolutionnary techniques to demonstrates that this type of blending is able to produce a self-organised system that relies on simple and general rules.

Obtained results

- An advanced visual module was produced for the simulation framework.
- An advanced physical model of the robots was produced inside the simulation framework.
- Relevant sensors and actuators were modeled for the simulation framework.
- A small working behaviour toolkit was produced for the foot-bots, using a new methodology in controller organisation.

Future work

Swarmanoid sensors and actuators should get their implementation validated with real data when it becomes available. Also, some speed optimisations on the simulator are required

on the scenegraph, using smart structures such as kd-trees with amortised and lazy neighbour search to allow a fast search of the swarmanoid Space when dealing with big numbers of robots. Finally, basic Push-Pull controllers should be adapted to circumvent hardware changes. Later on, a full scale parametric exploration should be done to achieve better performances of the controllers.

Bibliography

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

Arnaud, R. and Barnes, M. C. (2006). *Collada: Sailing the Gulf of 3d Digital Content Creation*. AK Peters Ltd.

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., R.Franks, N., 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.

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.

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)*.

Jong, K. A. D. (2002). *Evolutionary Computation*. The MIT Press.

Junker, G. (2006). *Pro OGRE 3D Programming*. Apress, Berkely, CA, USA.

Karn, P. (1990). Maca - a new channel access method for packet radio. In *ARRL 9th Computer Networking Conference*.

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

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., Floreano, D., Deneubourg, J.-L., Nolfi, S., Gambardella, L., and Dorigo, M. (2004). Swarm-bot: A new distributed robotic concept. *Autonomous Robots*, 17(2–3):193–221.

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

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.

Seyfried, J., Szymanski, M., Bender, N., Estana, R., Thiel, M., and Wörn, H. (2005). The i-swarm project: Intelligent small world autonomous robots for micro-manipulation. *Swarm Robotics, LNCS 3342, Springer*, pages 70–83.

Appendix

.1 3D object loading

Listing 1: XML DTD Format for 3D model importation.

```
<!ELEMENT scene ( nodes?, externals?, environment?, terrain?, userDataReference?,
 octree?, light?, camera? ) >
<!ATTLIST scene
formatVersion CDATA #REQUIRED
id      ID    #IMPLIED
sceneManager CDATA #IMPLIED
minOgreVersion CDATA #IMPLIED
author    CDATA #IMPLIED
>

<!ELEMENT terrain EMPTY>
<!ATTLIST terrain
dataFile CDATA #IMPLIED
>

<!ELEMENT nodes ( node*, position?, rotation?, scale? ) >

<!ELEMENT node ( position?, rotation?, scale?, lookTarget?, trackTarget?, node*, entity*,
 light*, camera*, particleSystem*, billboardSet*, plane*, userDataReference? ) >
<!ATTLIST node
name   CDATA #IMPLIED
id     ID    #IMPLIED
isTarget ( true | false ) "true"
>

<!ELEMENT particleSystem ( userDataReference? ) >
<!ATTLIST particleSystem
name CDATA #IMPLIED
id   ID    #IMPLIED
file CDATA #REQUIRED
>

<!ELEMENT light ( position?, normal?, colourDiffuse?, colourSpecular?, lightRange?,
lightAttenuation?, userDataReference? ) >
<!ATTLIST light
name    CDATA #IMPLIED
id      ID    #IMPLIED
type ( point | directional | spot | radPoint ) "point"
visible ( true | false ) "true"
castShadows ( true | false ) "true"
>

<!ELEMENT camera ( clipping?, position?, rotation?, normal?, lookTarget?, trackTarget?,
 userDataReference? ) >
<!ATTLIST camera
name    CDATA #IMPLIED
id      ID    #IMPLIED
fov     CDATA #IMPLIED
aspectRatio  CDATA #IMPLIED
projectionType ( perspective | orthographic ) "perspective"
>
```

```
<!ELEMENT trackTarget ( localDirection?, offset? ) >
<!ATTLIST trackTarget
nodeName CDATA #REQUIRED
>

<!ELEMENT lookTarget ( position?, localDirection? ) >
<!ATTLIST lookTarget
nodeName CDATA #IMPLIED
relativeTo ( local | parent | world )
>

<!ELEMENT lightAttenuation EMPTY>
<!ATTLIST lightAttenuation
range CDATA #IMPLIED
constant CDATA #IMPLIED
linear CDATA #IMPLIED
quadratic CDATA #IMPLIED
>

<!ELEMENT lightRange EMPTY>
<!ATTLIST lightRange
inner CDATA #REQUIRED
outer CDATA #REQUIRED
falloff CDATA #REQUIRED
>

<!ELEMENT entity ( vertexBuffer?, indexBuffer?, userDataReference? ) >
<!ATTLIST entity
name CDATA #IMPLIED
id ID #IMPLIED
meshFile CDATA #REQUIRED
materialFile CDATA #IMPLIED
static ( true | false ) "false"
castShadows ( true | false ) "true"
>

<!ELEMENT environment ( fog?, skyBox?, skyDome?, skyPlane?, clipping?, colourAmbient?,
colourBackground?, userDataReference? ) >

<!ELEMENT clipping EMPTY>
<!ATTLIST clipping
near CDATA #REQUIRED
far CDATA #REQUIRED
>

<!ELEMENT fog ( colourDiffuse? ) >
<!ATTLIST fog
expDensity CDATA #DEFAULT "0.001"
linearStart CDATA #DEFAULT "0.0"
linearEnd CDATA #DEFAULT "1.0"
mode ( none | exp | exp2 | linear ) "none"
>

<!ELEMENT skyBox ( rotation? ) >
<!ATTLIST skyBox
material CDATA #REQUIRED
distance CDATA #DEFAULT "5000"
drawFirst ( true | false ) "true"
>

<!ELEMENT skyDome ( rotation? ) >
<!ATTLIST skyDome
material CDATA #REQUIRED
curvature CDATA #DEFAULT "10"
tiling CDATA #DEFAULT "8"
distance CDATA #DEFAULT "4000"
drawFirst ( true | false ) "true"
>

<!ELEMENT skyPlane EMPTY>
<!ATTLIST skyPlane
material CDATA #REQUIRED
planeX CDATA #DEFAULT "0"
```

```
planeY   CDATA #DEFAULT "-1"
planeZ   CDATA #DEFAULT "0"
planeD   CDATA #DEFAULT "5000"
scale    CDATA #DEFAULT "1000"
bow      CDATA #DEFAULT "0"
tiling   CDATA #DEFAULT "10"
drawFirst ( true | false )  "true"
>

<!ELEMENT billboardSet ( billboard* ) >
<!ATTLIST billboardSet
name   CDATA #REQUIRED
material CDATA #REQUIRED
id     ID   #IMPLIED
width   CDATA #DEFAULT "10"
height   CDATA #DEFAULT "10"
type ( orientedCommon | orientedSelf | point ) "point"
origin ( bottomLeft | bottomCenter | bottomRight | left | center | right | topLeft |
topCenter | topRight ) "center"
>

<!ELEMENT billboard ( position?, rotation?, colourDiffuse? ) >
<!ATTLIST billboard
id ID #IMPLIED
width CDATA #IMPLIED
height CDATA #IMPLIED
>

<!ELEMENT plane ( normal, upVector?, vertexBuffer?, indexBuffer? ) >
<!ATTLIST plane
name     CDATA #REQUIRED
id       ID   #IMPLIED
distance   CDATA #REQUIRED
width    CDATA #REQUIRED
height     CDATA #REQUIRED
xSegments   CDATA #DEFAULT "1"
ySegments   CDATA #DEFAULT "1"
numTexCoordSets CDATA #DEFAULT "1"
uTile    CDATA #DEFAULT "1"
vTile    CDATA #DEFAULT "1"
material   CDATA #IMPLIED
normals ( true | false )  "true"
>

<!ELEMENT vertexBuffer EMPTY>
<!ATTLIST vertexBuffer
usage ( static | dynamic | writeOnly | staticWriteOnly | dynamicWriteOnly )
"staticWriteOnly" useShadow ( true | false ) "true"
>

<!ELEMENT indexBuffer EMPTY>
<!ATTLIST indexBuffer
usage ( static | dynamic | writeOnly | staticWriteOnly | dynamicWriteOnly )
"staticWriteOnly" useShadow ( true | false ) "true"
>

<!ELEMENT externals ( item* ) >

<!ELEMENT item ( file ) >
<!ATTLIST item
type CDATA #REQUIRED
>

<!ELEMENT file EMPTY>
<!ATTLIST file
name CDATA #REQUIRED
>

<!ELEMENT position EMPTY>
<!ATTLIST position
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>
```

```
<!ELEMENT rotation EMPTY>
<!ATTLIST rotation
qx CDATA #IMPLIED
qy CDATA #IMPLIED
qz CDATA #IMPLIED
qw CDATA #IMPLIED
axisX CDATA #IMPLIED
axisY CDATA #IMPLIED
axisZ CDATA #IMPLIED
angle CDATA #IMPLIED
angleX CDATA #IMPLIED
angleY CDATA #IMPLIED
angleZ CDATA #IMPLIED
>

<!ELEMENT normal EMPTY>
<!ATTLIST normal
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>

<!ELEMENT upVector EMPTY>
<!ATTLIST upVector
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>

<!ELEMENT offset EMPTY>
<!ATTLIST offset
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>

<!ELEMENT localDirection EMPTY>
<!ATTLIST localDirection
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>

<!ELEMENT scale EMPTY>
<!ATTLIST scale
x CDATA #REQUIRED
y CDATA #REQUIRED
z CDATA #REQUIRED
>

<!ELEMENT colourDiffuse EMPTY>
<!ATTLIST colourDiffuse
r CDATA #REQUIRED
g CDATA #REQUIRED
b CDATA #REQUIRED
>

<!ELEMENT colourSpecular EMPTY>
<!ATTLIST colourSpecular
r CDATA #REQUIRED
g CDATA #REQUIRED
b CDATA #REQUIRED
>

<!ELEMENT colourAmbient EMPTY>
<!ATTLIST colourAmbient
r CDATA #REQUIRED
g CDATA #REQUIRED
b CDATA #REQUIRED
>

<!ELEMENT colourBackground EMPTY>
<!ATTLIST colourBackground
```

```
r CDATA #REQUIRED
g CDATA #REQUIRED
b CDATA #REQUIRED
>

<!ELEMENT userDataReference EMPTY>
<!ATTLIST userDataReference
id CDATA #REQUIRED
>

<!ELEMENT octree ( octnode ) >
<!ATTLIST octree
binFile CDATA #REQUIRED
>

<!ELEMENT octNode ( octNode*, octMesh* ) >
<!ATTLIST octNode
px CDATA #REQUIRED
py CDATA #REQUIRED
pz CDATA #REQUIRED
width CDATA #REQUIRED
height CDATA #REQUIRED
depth CDATA #REQUIRED
>

<!ELEMENT octMesh ( octGeometry, octMaterial ) >

<!ELEMENT octGeometry EMPTY>
<!ATTLIST octGeometry
binaryDataOffset CDATA #REQUIRED
vertTotal CDATA #REQUIRED
triTotal CDATA #REQUIRED
normalTotal CDATA #IMPLIED
colorTotal CDATA #IMPLIED
texSets CDATA #IMPLIED
texTotal CDATA #IMPLIED
>

<!ELEMENT octMaterial EMPTY>
<!ATTLIST octMaterial
name CDATA #REQUIRED
texture CDATA #IMPLIED
>
```

Listing 2: Example of XML code for modular entity parsing.

```
<node name="HB_right_hand">
<position x="-63.943810" y="-0.904891" z="-0.276499"/>
<rotation qx="0.078459" qy="-0.000000" qz="0.000000" qw="0.996917"/>
<scale x="1.000000" y="1.000000" z="1.000000"/>
<entity name="HB_right_hand" meshFile="HB_right_hand.mesh"/>
        <node name="HB_LFRH">
                <position x="-59.022900" y="-23.205893" z="23.109951"/>
                        <rotation qx="-0.000000" qy="0.000000" qz="0.000000" qw="1.000000"/>
                        <scale x="1.000000" y="1.000000" z="1.000000"/>
                        <entity name="HB_LFRH" meshFile="HB_LFRH.mesh"/>
                </node>
                <node name="HB_RFRH">
                        <position x="-59.022888" y="27.041294" z="-20.173742"/>
                        <rotation qx="-0.000000" qy="0.000000" qz="0.000000" qw="1.000000"/>
                        <scale x="1.000000" y="1.000000" z="1.000000"/>
                        <entity name="HB_RFRH" meshFile="HB_RFRH.mesh"/>
                </node>
        </node>
</node>
```

.2 3D module architecture

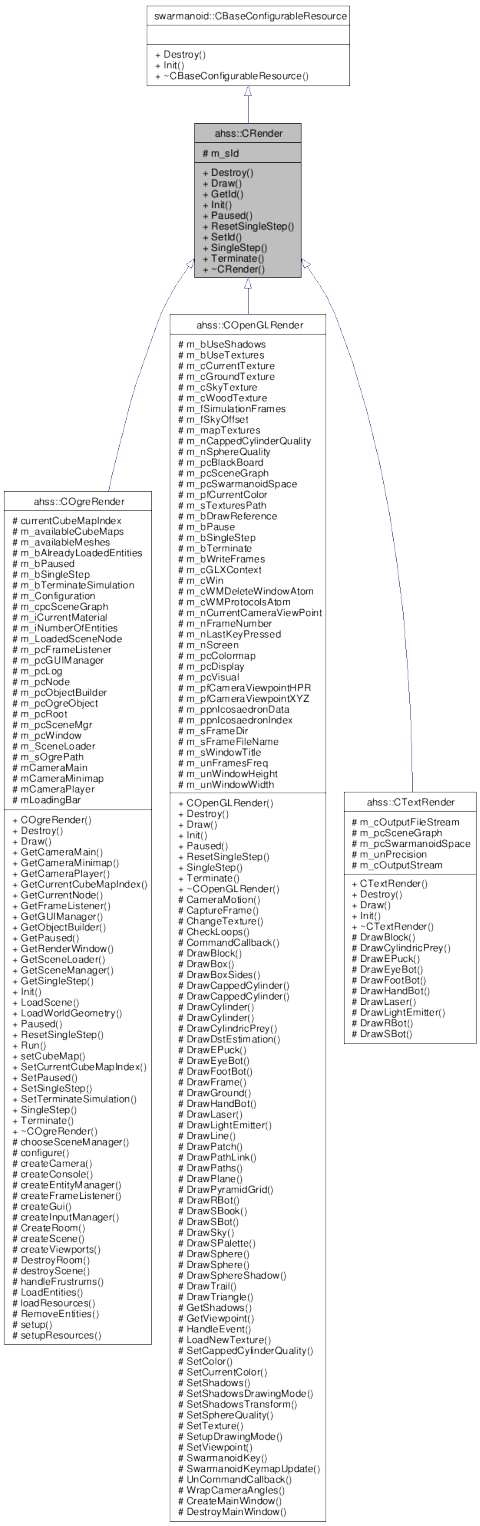
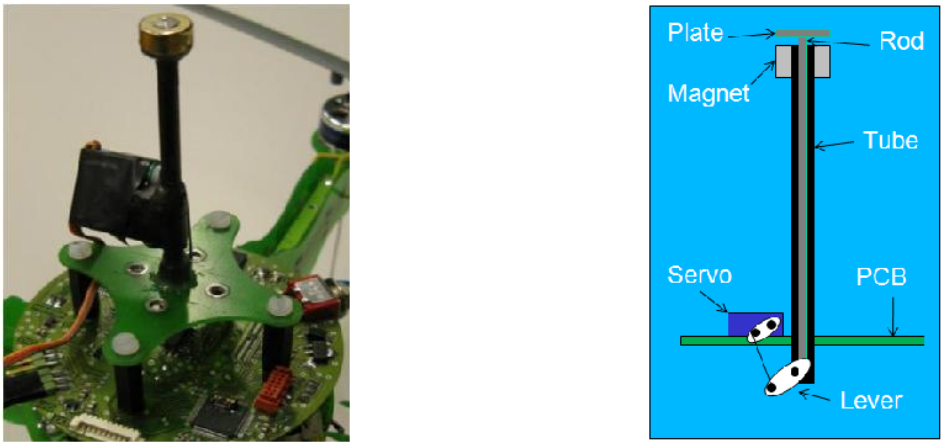


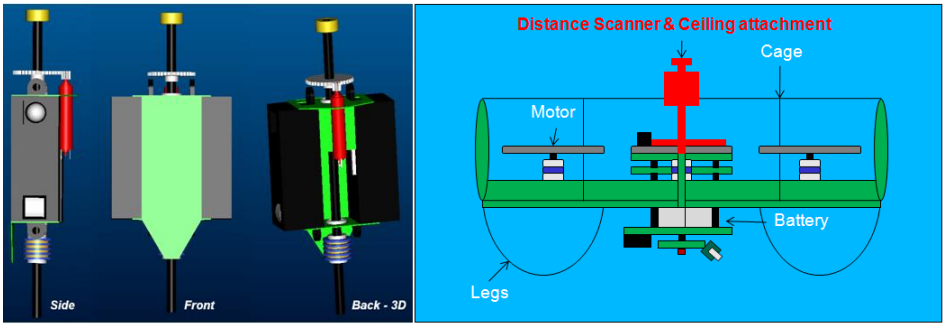
Figure 1: The Rendering SubClasses present inside Argos.

Figure 2: The COgreRender collaboration diagram.

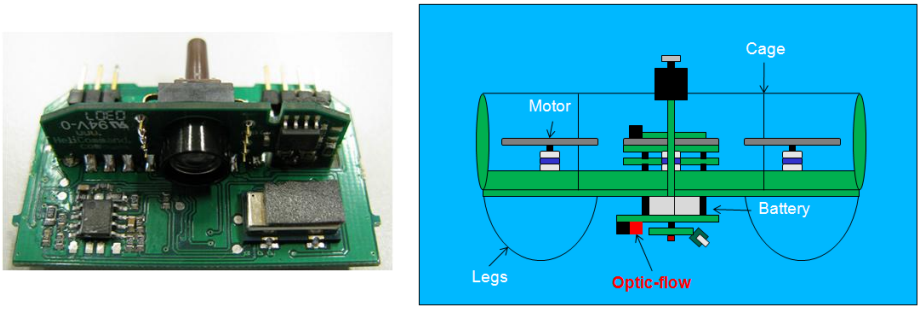
.3 Actuators and sensors



(a) Left: prototype ceiling attachment device, right: mechanical drawing.



(b) Omni-directional distance scanner: left - prototype model, right - mounting location.



(c) Optical flow sensor: left - optical sensor, right - mounting location.

Figure 3: Unimplemented eye-bot sensors.