ARGoS Framework Documentation

Carlo Pinciroli

<cpinciro@ulb.ac.be>

Eliseo Ferrante

<eferrante@ulb.ac.be>

Nithin Mathews

<nmathews@ulb.ac.be>

Arne Brutschy

<arne.brutschy@ulb.ac.be>

Revision History		
Revision 0.2	20.04.3009	AB
Added new structure for development guidelines		
Revision 0.1	26.03.3009	NM
Added Behavioral Toolkit		
Revision 0.0	25.03.2009	CP
DEA Transfer		

Table of Contents

1. Development

    - Distributed Development Model
    - High Level Coding Guidelines
    - Low Level Coding Guidelines

        - Unified Coding Style: Common Rules
        - Coding Recommendations

2. Installation

    - Obtaining ARGoS
    - Required libs and packages
    - Optional libs and packages
    - How to compile
    - Testing the Enviroment
    - Troubleshooting

3. Architecture of the ARGoS Framework

    - Architecture
    - Code Organization
    - Common Interface

        - Robot
        - Controller
        - Sensors and Actuators

    - Swarmanoid Space
    - Physics Engines

        - 2D Kinematic physics engine
        - 2D Dynamic physics engine
        - 3D Dynamic physics engine

    - Visualizations

        - Text-based visualization
        - Graphical visualization: OpenGL
        - Graphical visualization: OGRE

4. ARGoS in Practice

    - User Environment

        - Contents of the User Directory
        - User Scripts

    - Build Environment

        - Package Dependencies
        - Autotools Configuration
        - Build Scripts
        - Building for the Real Robots

    - Working with the Distributed Development Model

Writing a New Controller
Writing a New Behavior Controller

Motivation
Main idea
Implementation in ARGoS
How to use it?
Using the FSM

Definition of an Experiment
A Sample Experiment

List of Figures

3.1. Overall architecture of the Swarmanoid Simulator.
3.2. The package diagram of the Swarmanoid Simulator.
3.3. The class diagram of the core classes of the Swarmanoid Simulator.
3.4. A schematic class diagram of the Common Interface
3.5. The Swarmanoid Space and its global 3D coordinate space.
3.6. A schematic class diagram of the Swarmanoid Space.
3.7. The class diagram of the physics engine interface .
3.8. How planes can be placed in the Swarmanoid Space when using the 2D kinematics engine.
3.9. 2D kinematic collision models.
3.10. Visualization code class diagram.
3.11. Sample output of the basic text visualization.
3.12. OpenGL visualization example
3.13. OGRE visualization example
4.1. The main idea work-flow schema of the behavioral toolkit.
4.2. The behavioral toolkit with parallel behaviors.
4.3. Class diagram showing the implementation of the Behavioral Toolkit.
4.4. The initial setup of the experimental arena.
4.5. The behavior diagram of the footbot controller.
4.6. The behavior diagram of the eyebot controller.
4.7. The eyebot waiting for a footbot to grip the prey.
4.8. The footbot gripping the prey.
4.9. The eyebot driving the footbot to the nest.
4.10. The prey is brought to the nest.

Chapter 1. Development

Table of Contents

Distributed Development Model
High Level Coding Guidelines
Low Level Coding Guidelines

Unified Coding Style: Common Rules
Coding Recommendations

Distributed Development Model

This has to be filled by Rehan.

High Level Coding Guidelines

This has to be filled by Carlo.

Low Level Coding Guidelines

The use of the following rules is mandatory for all software implementation work. Changes that don’t comply with these rules will not be accepted by the software coordinator.

Unified Coding Style: Common Rules

Rule 1: Indentation

All code blocks that are included into braces have to be indented by two (2) space characters.

Example:

```
void function ()
{
  code here;
}
```

Rule 2: Braces (not obligatory)

The opening brace is placed on a new line on the same indentation level as the defining keyword (e.g. void, if, for, while, etc.). The included code block starts at the following line and is intended (see Rule 1). The closing brace is placed on the same indentation level as the opening brace.

Example:

```
void function ()
{
```

```
    if (<expression>)
    {
      <code here>
    }
  }
```

To be continued...

Coding Recommendations

To be continued...

Chapter 2. Installation

Table of Contents

Obtaining ARGoS
Required libs and packages
Optional libs and packages
How to compile
Testing the Enviroment
Troubleshooting

Obtaining ARGoS

ARGoS is the *Autonomous Robots Go Swarming* Simulator designed for the Swarmanoid Project. ARGoS can be dowloaded only from the SVN Repository hosted at the IRIDIA lab. To obtain access to it ask Rehan O'Grady (<rehanog@gmail.com>). Once you are provided with a user name, type the following command at the prompt:

```
svn checkout https://USERNAME@iridia-dev.ulb.ac.be/projects/argos/svn
```

Required libs and packages

..

Optional libs and packages

..

How to compile

..

Testing the Enviroment

..

Troubleshooting

..

Chapter 3. Architecture of the ARGoS Framework

Table of Contents

Architecture
Code Organization
Common Interface

- Robot
- Controller
- Sensors and Actuators

Swarmanoid Space
Physics Engines

- 2D Kinematic physics engine
- 2D Dynamic physics engine
- 3D Dynamic physics engine

Visualizations

- Text-based visualization
- Graphical visualization: OpenGL
- Graphical visualization: OGRE

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

Architecture

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

1. Initialize Simulation

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

2. Arena Visualization

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

3. Run Controllers

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

4. Physics Update

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

5. Next Simulation Step

    Return to point 2.

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

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

Figure 3.1. Overall architecture of the Swarmanoid Simulator.

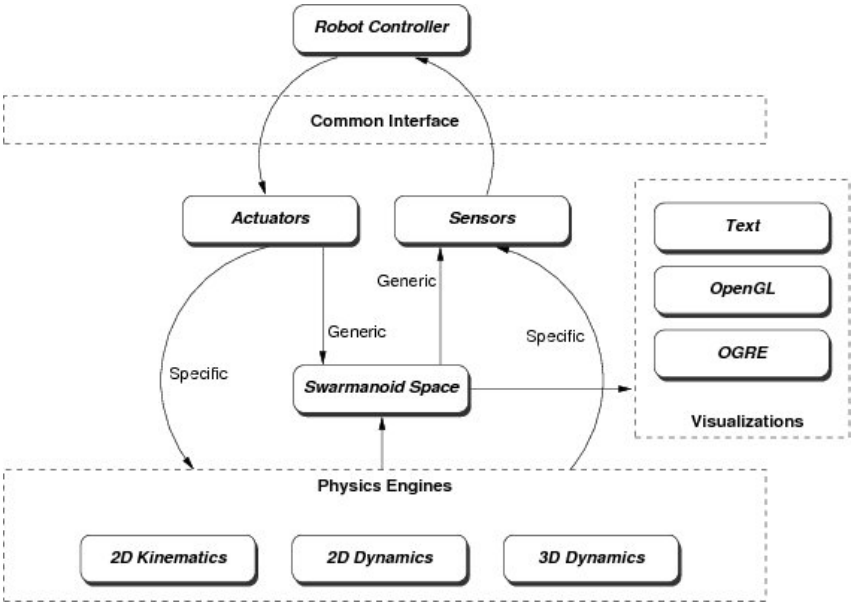


Figure ?? depicts the architecture of the Swarmanoid Simulator. The 3D Swarmanoid Space can be found at the center of the picture. Physics engines access it to update its status and visualizations display it.

Robots interact with the environment through simulated sensors and actuators. As already discussed in Chapter ??, the ease to transfer controllers developed in simulation to real robots heavily depends on how sensors and actuators have been modeled. The architecture of the Swarmanoid Simulator does not favor any approach (physics-based or sample-based), thus letting the developer choose the best approach in each case.

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

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

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

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

1. Initialize Simulation

    1-1

        Create the simulated arena

1-2

Place all the simulated objects in their initial positions in the Swarmanoid Space

1-3

Create robot controllers

1-4

Initialize physics engines and assign robots to them

1-5

Initialize visualizations

2. Arena Visualization

(for each visualisation)

2-1

Display the status of the Swarmanoid Space.

3. Physics Update

(for each physics engine)

3-1

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

3-2

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

4. Next Simulation Step

4-1

Return to point 2.

Code Organization

Figure 3.2. The package diagram of the Swarmanoid Simulator.

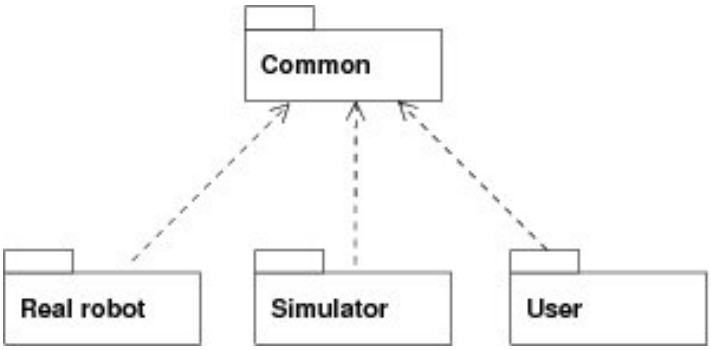
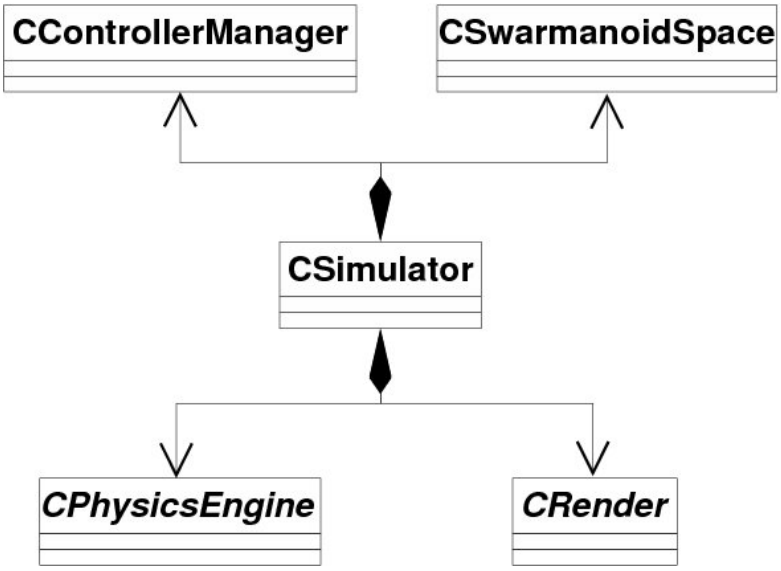


Figure reports the software packages in which the code has been divided and their mutual dependencies. The division in packages has been designed with the intentions of maximizing code reuse while keeping the functionalities logically separated.

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

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

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



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

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

Common Interface

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

Figure 3.4. A schematic class diagram of the Common Interface

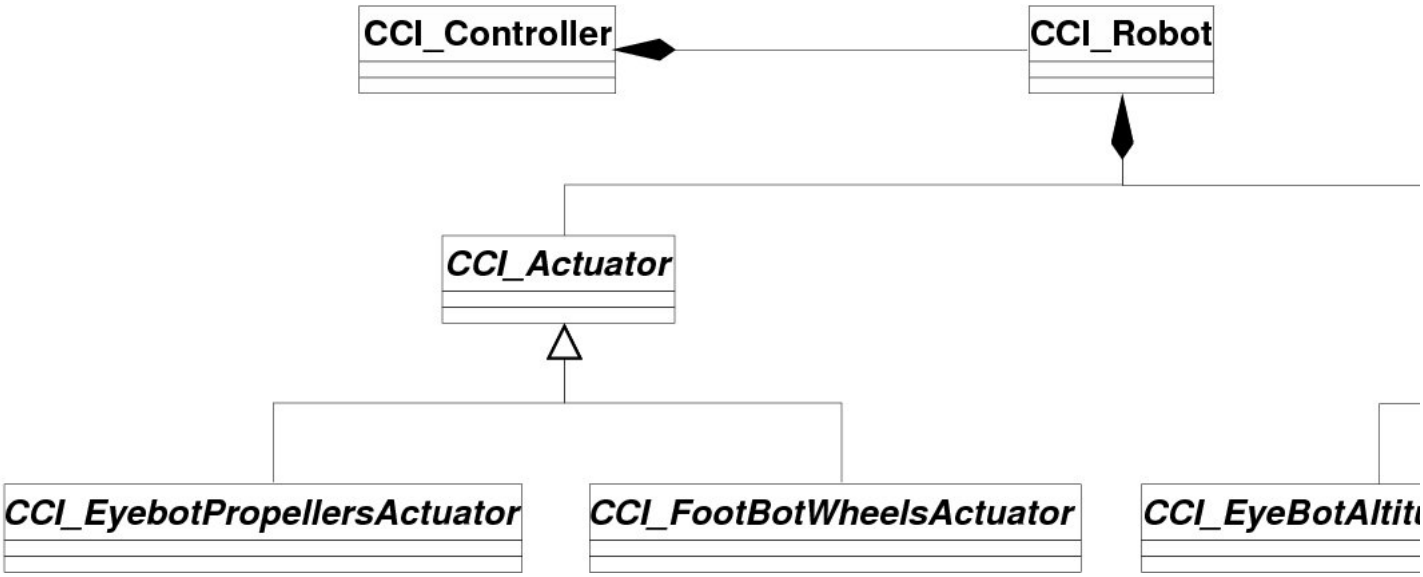


Figure presents a schematic class diagram of the Common Interface, which encompasses a robot interface, a controller interface and sensors and actuators interface. The Common Interface provides virtual access to all the features necessary to develop a controller in a transparent way with respect to the simulated or the real world. In the following, we briefly describe each component of the Common Interface.

Robot

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

Controller

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

The controller interface basically provides three main methods: Init, ControlStep and Destroy. Every user-defined controller must implement these functions, which are executed in the same way on the simulated and on the real robots.

Sensors and Actuators

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

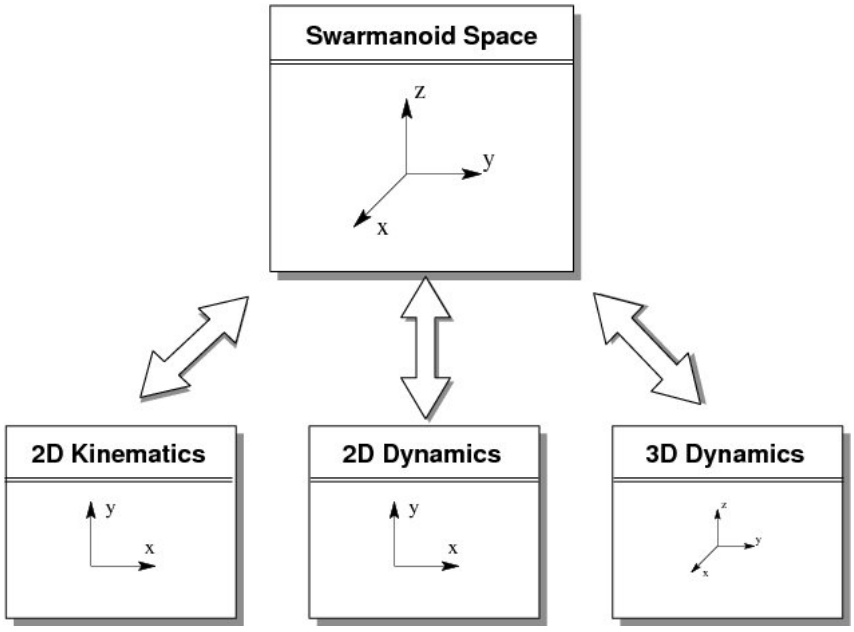
From the general interfaces, we derive the detailed interface for each particular device. In Figure only some example interfaces are displayed, such as the one for the footbot wheels or the one for the eyebot altitude sensor. At this level, each interface corresponds to a particular device. We will also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

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

Swarmanoid Space

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

Figure 3.5. The Swarmanoid Space and its global 3D coordinate space.



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

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

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

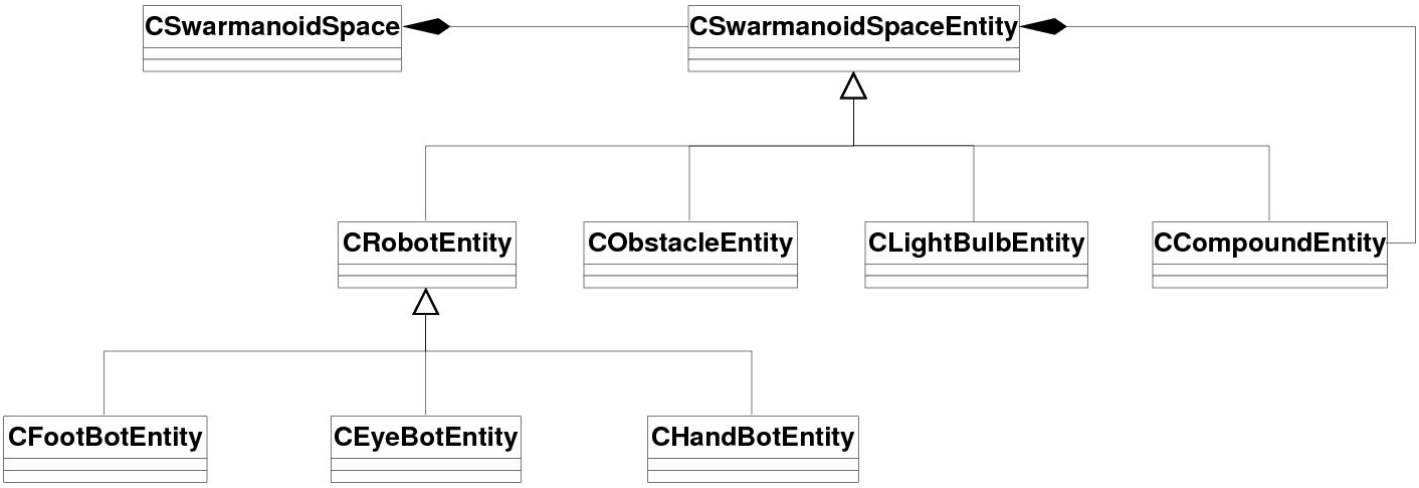
Furthermore, generic sensors and actuators are allowed to access information in the Swarmanoid Space. The proximity sensors, for example, are provided functions to retrieve all the entities with a certain distance from a point in the space or an entity. Other functions have been implemented to check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

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

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

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

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



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

CRobotEntity adds to the base class the association to an object of type CCI_Controller, which is explained in Section . CFootBotEntity, CHandBotEntity and CEyeBotEntity are an extension of CRobotEntity, which store the specific data of each robot, such as the LEDs color and the gripper aperture status for the footbot.

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

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

Physics Engines

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

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

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

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

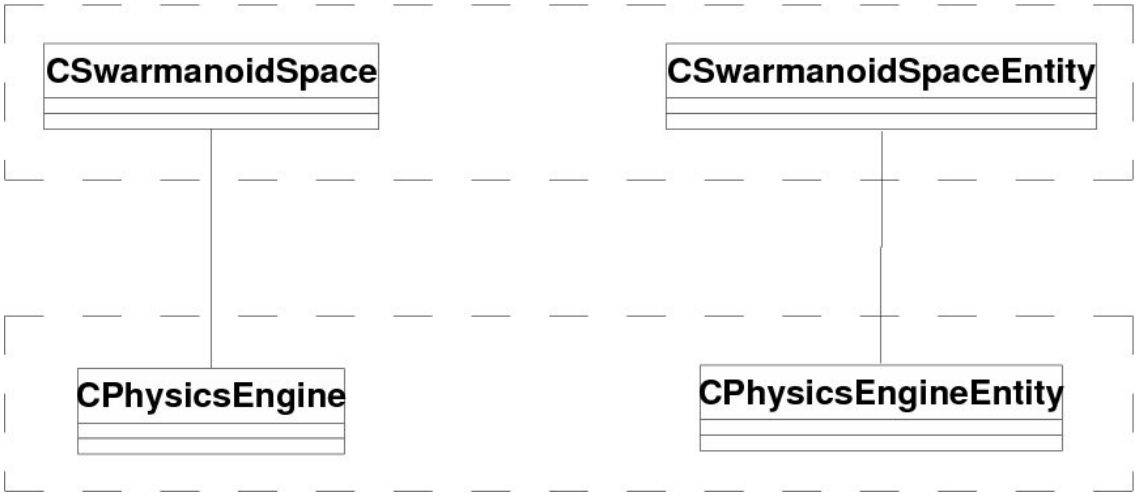
There exists a number of different ways to represent and implement motion equations, friction forces, collision detection, and equation integration, resulting in a number of different physics engines.

The main novel aspect of the architecture of the Swarmanoid Simulator is the possibility to implement physics engines as plugins and run them in parallel. The degree of flexibility and generality deriving from such design choice allows the user to decide which physics engine to use to simulate a set of robots, so that experiments can be optimized and accuracy is ensured where it is really needed.

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

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

Figure 3.7. The class diagram of the physics engine interface .



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

The first, CPhysicsEngine, defines the main functions of an engine, that basically are concentrated in the Update method. At each time step, the simulator calls this method to ask the engine to calculate new positions and headings for the entities assigned to it. The association with CSwarmanoidSpace allows the physics engine to retrieve useful information from the scene graphs, if needed.

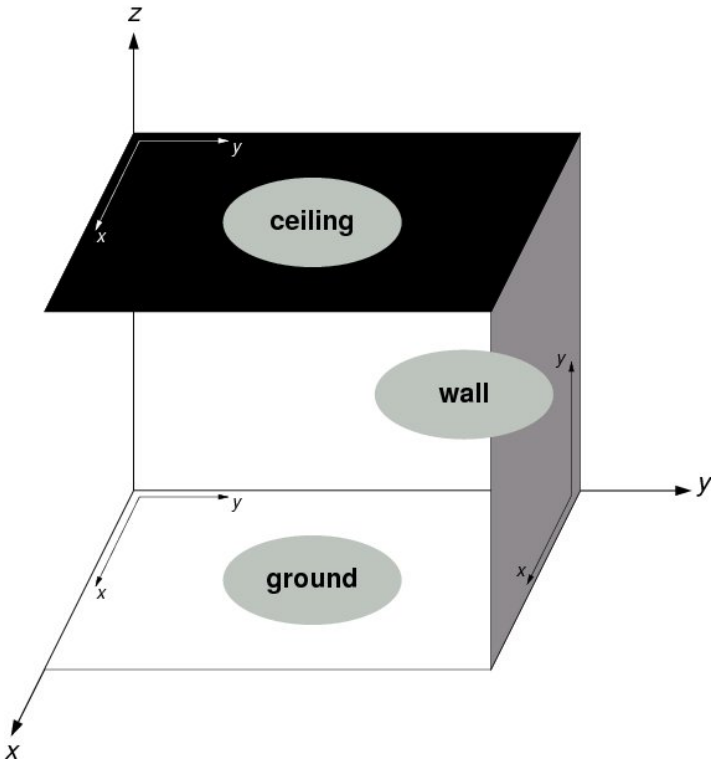
On the other hand, the role of CPhysicsEngineEntity is to provide a basic definition of the entity as seen inside a physics engine. This interface has been designed to be absolutely minimal to keep the constraints it imposes on the development of a physics engine null or negligible. For this reason,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

2D Kinematic physics engine

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

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

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



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

Figure 3.9. 2D kinematic collision models.

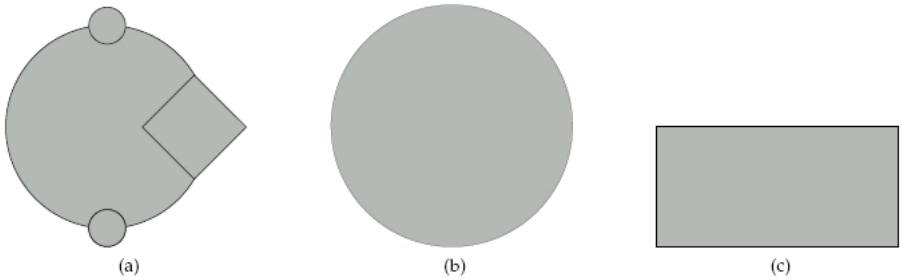


Figure depicts the simple geometrical models for collision detection. Footbots are composed by a big central circle (the body), two smaller circles (the wheels) and a square (the gripper). The model of an eyebot is even more essential: just a circle. Eventually, handbots do not have a defined stable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

Gripping in this engine is supported in a very basic way. Footbots are provided a gripper that can only attach to target objects termed *preys* which possess a special gripping strip around their bodies. Furthermore, footbots cannot grip other footbots. This is a simplistic approach to maintain performance good. Also, rigid body dynamics require masses and accelerations to be simulated properly, but, as we have already illustrated, in a kinematic physics engine acceleration and masses are not taken into account.

2D Dynamic physics engine

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

3D Dynamic physics engine

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

For eyebots, an ode based physics engine supporting the prototype robot is already available in the omiss simulator[2], though it has not been integrated in the architecture yet.

Visualizations

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

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

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

The modular architecture of the Swarmanoid Simulator has been designed to let the developer insert visualizations as plugins in the same way sensors, actuators and physics engines are treated. Similarly, a software interface named CRender has been kept as thin as possible to leave the developer free to define the most appropriate visualization logic.

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

Figure 3.10. Visualization code class diagram.

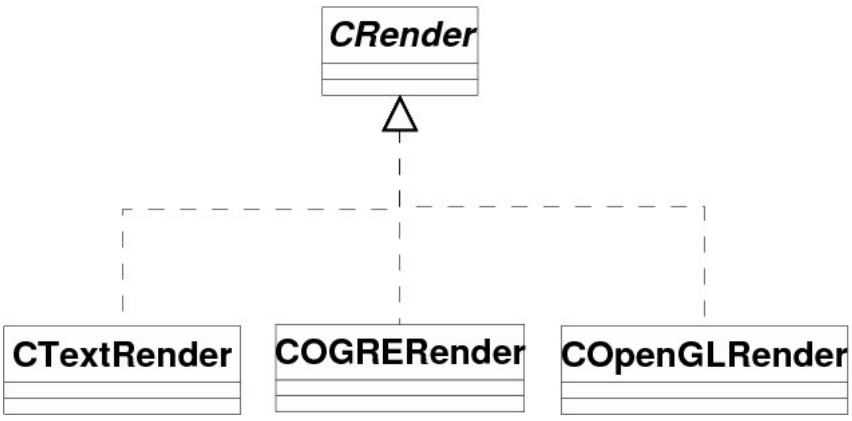


Figure displays the class diagram of the visualization code. In the following sections we will present these modules.

Text-based visualization

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

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

Figure 3.11. Sample output of the basic text visualization.

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

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

clock

The current clock tick count. The actual duration of the clock tick in milliseconds is specified in the xml file as described in Section ??.

Entity type

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

Entity id

The identificative name of the entity.

X, Y, Z

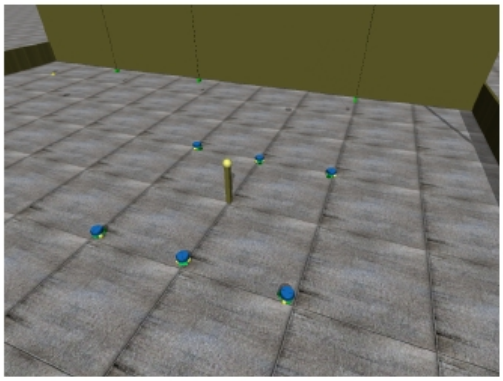
The position of the entity in the Swarmanoid Space.

Alpha, Beta, Gamma

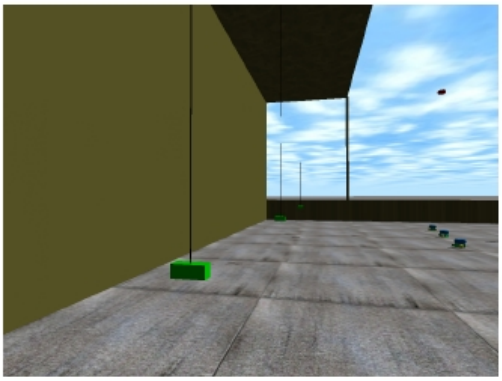
The value of the Euler angles storing the orientation of the entity expressed in degrees.

Graphical visualization: OpenGL

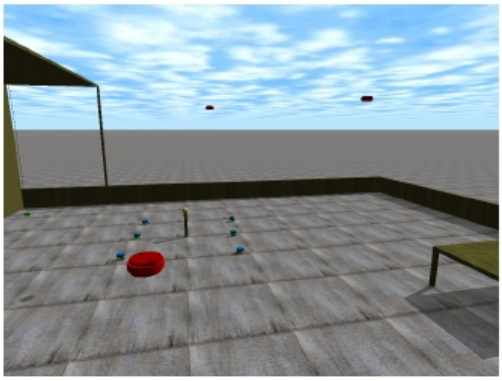
Figure 3.12. OpenGL visualization example



(a)



(b)



(c)

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

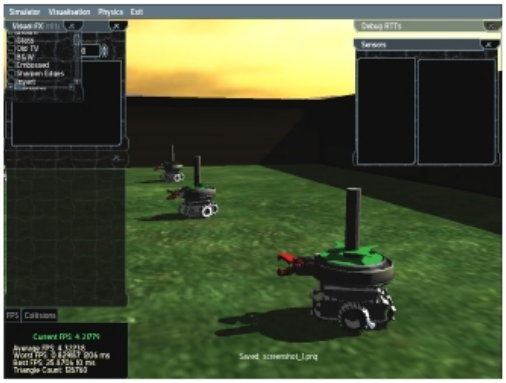
The OpenGL renderer provides a simple three-dimensional visualization of the simulated arena. Code is based on OpenGL, as the name suggests, a fast library for 3D rendering used in many different fields, from modeling to video games. The internal code structure is purposely straightforward and the robot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

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

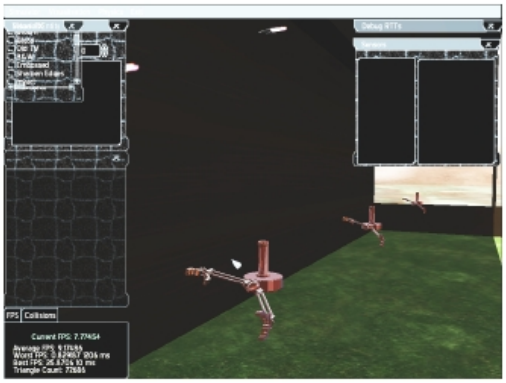
Some sample frames taken with this visualization are displayed in Figure . Although code structure is simple, this module provides a good number of useful functions. For instance, it is possible to move the camera by dragging the mouse and to record the current camera position. The user can store up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

Graphical visualization: OGRE

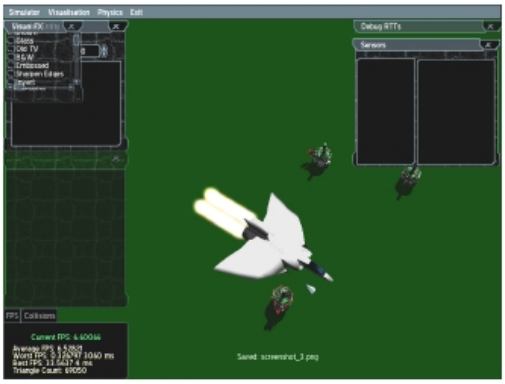
Figure 3.13. OGRE visualization example



(a)



(b)



(c)

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

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

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

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

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

---

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

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

Chapter 4. ARGoS in Practice

Table of Contents

User Environment

- Contents of the User Directory
- User Scripts

Build Environment

- Package Dependencies
- Autotools Configuration
- Build Scripts
- Building for the Real Robots

Working with the Distributed Development Model
Writing a New Controller
Writing a New Behavior Controller

- Motivation
- Main idea
- Implementation in ARGoS
- How to use it?
- Using the FSM

Definition of an Experiment
A Sample Experiment

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

User Environment

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

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

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

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

Contents of the User Directory

The user directory contains the following files and directories:

configure.ac

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

Makefile.am

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

bootstrap.sh

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

build.sh

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

simulation_main.cpp

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

simulation_build

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

real_robot_build

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

controllers

: This directory contains all the controllers of the user.

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

User Scripts

To be written by Carlo/Arne/...

Build Environment

Package Dependencies

To be written by Eliseo/Arne...

Autotools Configuration

To be written by Eliseo/Arne...

Build Scripts

To be written by Eliseo/Arne...

Building for the Real Robots

To be written by Eliseo/Arne...

Working with the Distributed Development Model

To be written by Nithin...

Writing a New Controller

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

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

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

```
#ifndef _CSAMPLEFOOTBOTCONTROLLER_H_
#define _CSAMPLEFOOTBOTCONTROLLER_H_
#include "ci_controller.h"
#include "footbot/ci_footbot_wheels_actuator.h"
#include "config.h"
class CSampleFootBotController: public CCI_Controller {
    // Associations
    private:
        CCI_FootBotWheelsActuator* m_pcFootBotWheelsActuator;
    // Attributes
    // Operations
    public:
        virtual int Init ( const TConfigurationTree t_tree );
        virtual void ControlStep ( );
        virtual void Destroy ( );
};
#endif
```

The interface is very simple: we have two methods, called Init and Destroy, that respectively initialize and destroy the object. They do not coincide with the constructor and the destructor, because the construction/destruction logic is left to the framework to manage; anyway, from the point of view of the user, Init contains the code to retrieve references to sensors and actuators, to create structures such as neural networks, and so on. Furthermore, this method receives as input an xml data structure that is the configuration subtree specified for this controller in the configuration file (refer to Section ?? for details).

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

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

```
#include "sample_footbot_controller.h"
using namespace swarmanoid;
using namespace ahss;
using namespace std;

int CSampleFootBotController::Init(const TConfigurationTree t_tree)
{
    m_pcFootBotWheelsActuator = (CCI_FootBotWheelsActuator*)(GetRobot( )->GetActuator( "footbot_wheels" ));
    return CCI_Controller::RETURN_OK;
}
void CSampleFootBotController::ControlStep ( )
{
    m_pcFootBotWheelsActuator->SetFootBotWheelsAngularVelocity( 0.314, 0.628 );
}
void CSampleFootBotController::Destroy ( ){}
REGISTER_CONTROLLER( CSampleFootBotController,"sample_footbot_controller" )
```

As explained in Section ??, an object of type CCI_Robot is a container of sensors and actuators. Therefore in method Init a reference to the wheels actuator is obtained by means of method CCI_Robot::GetActuator. The string value passed to it is a type label associated with the actuator at the moment of registering it; as it will be shown in Section ??, such type label is also used in the xml configuration file to let the simulator initialize the actuator.

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

Writing a New Behavior Controller

This section presents another approach for writing robot controllers which leads to a more efficient development process. According to this approach, robot controllers and robot behaviors are two clearly distinguished concepts. The programmer will be focused on the development of behaviors only, whereas the controller will be just an empty container around them. Behaviors, which consists in controller's building blocks (modules), can be programmed in an abstract way, without the need to access directly to the robot's hardware. Furthermore, they can be tested/debugged separately or, most importantly, combined together in a arbitrary and general way.

Motivation

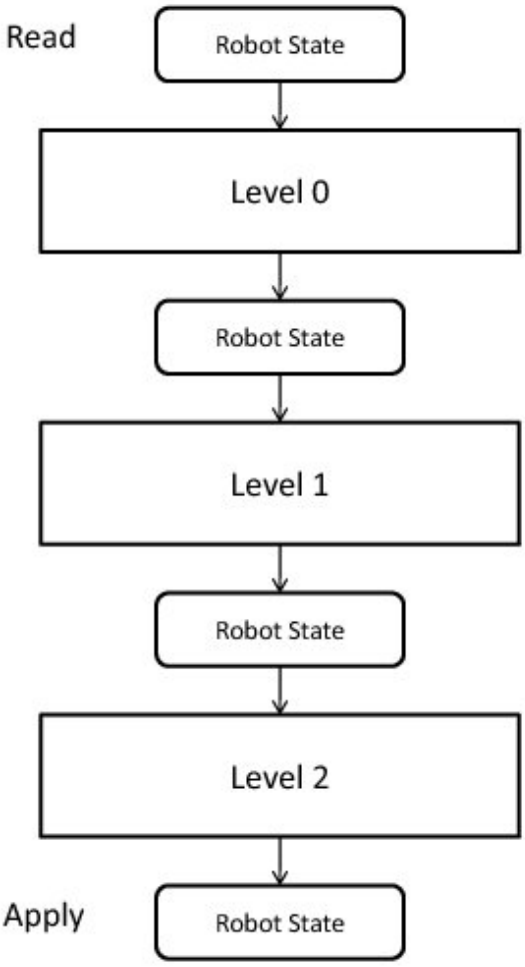
Developing controllers as explained in the section called “Writing a New Controller” can very quickly produce monolithic piece of code that will most likely not scale to more complex tasks. Code re-usability is in general not ensured since it is not clear how to identify self-contained modules or to re-use them in different controllers, except with the classical copy and paste technique which is error-prone and could lead to inefficiency on both small and long time scales. As a consequence, it is not easy to share code between different developers. The new behavioral toolkit architecture has been thought to overcome the above limitations. In fact, it has been developed by keeping in mind central concepts such us modularity and code re-usability. In the following sections, after describing the architecture implementation in ARGoS, we will demonstrate with examples how to write a behavior controller and which its advantages are.

Main idea

The main idea behind the behavioral toolkit is straightforward and it is summarized in Figure 4.1, “The main idea work-flow schema of the behavioral toolkit.” We define an object called robot state that is, as a matter of fact, a wrapper around the robot's sensors and actuators. The robot state is read once each timestep, before any behavior starts its execution. It is then injected into the first behavior that has, say, priority level 0. It can read the sensors variables from the robot state, do all its computation, and then write actuator's variables inside the same robot state. At this point, the state is injected inside a behavior with higher level of priority (1 in this case). This behavior will perceive the same values for all the sensors but it is allowed to write other values inside the actuators, hence possibly overriding whatever

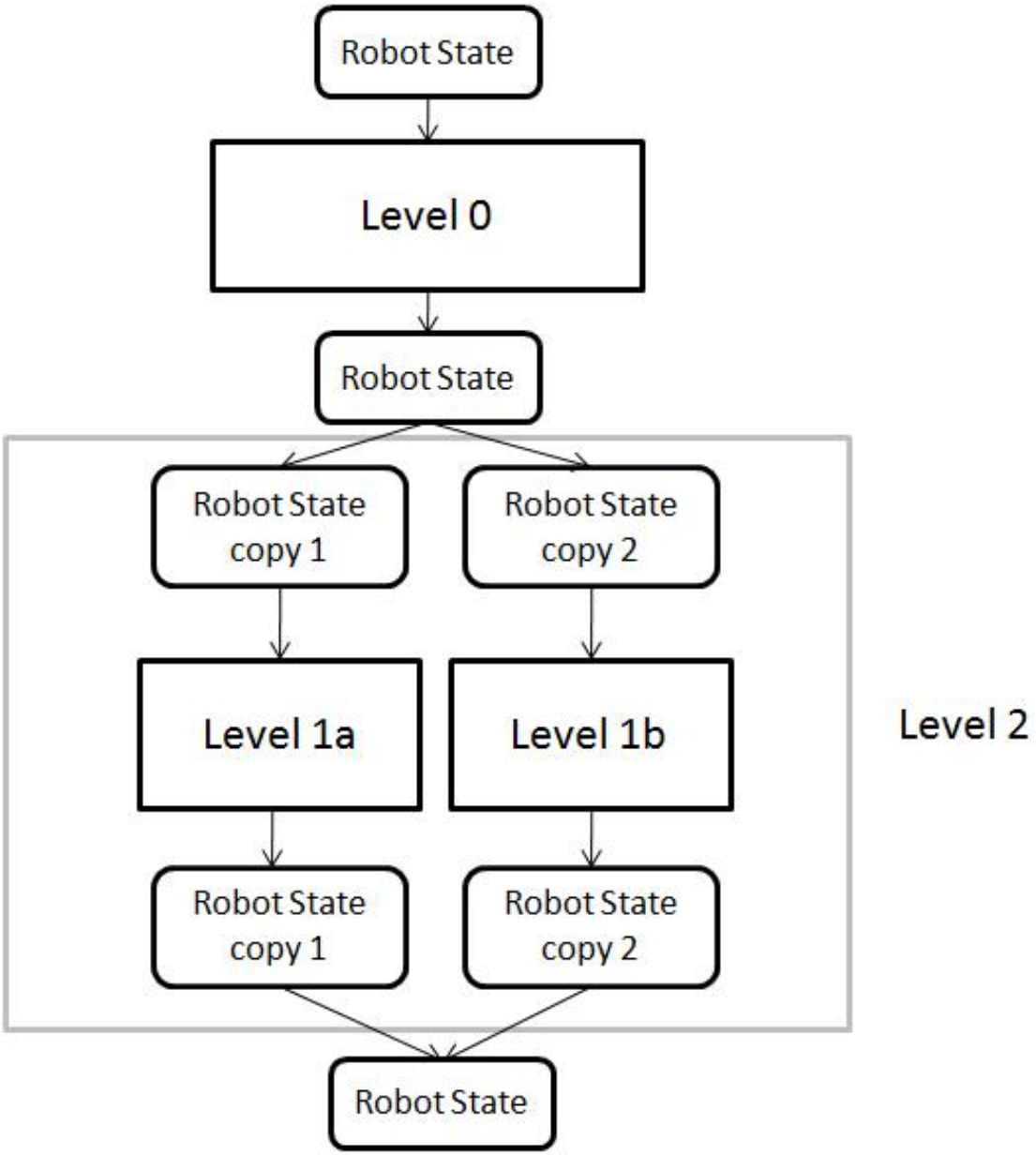
the previous behavior has written. This process can be repeated an arbitrary number of times. the section called “Writing a combiner behavior (sequentially)” shows a practical example explaining how this works in practice.

Figure 4.1. The main idea work-flow schema of the behavioral toolkit.



With the behavioral toolkit you can also write behaviors that are executed (virtually) concurrently and that can then be combined into a top behavior. This process is exemplified in Figure 4.2. “The behavioral toolkit with parallel behaviors.”. The key feature that enables this is the copy functionality of the robot state. Example of this will be provided in the section called “ Writing a combiner behavior (parallely)”

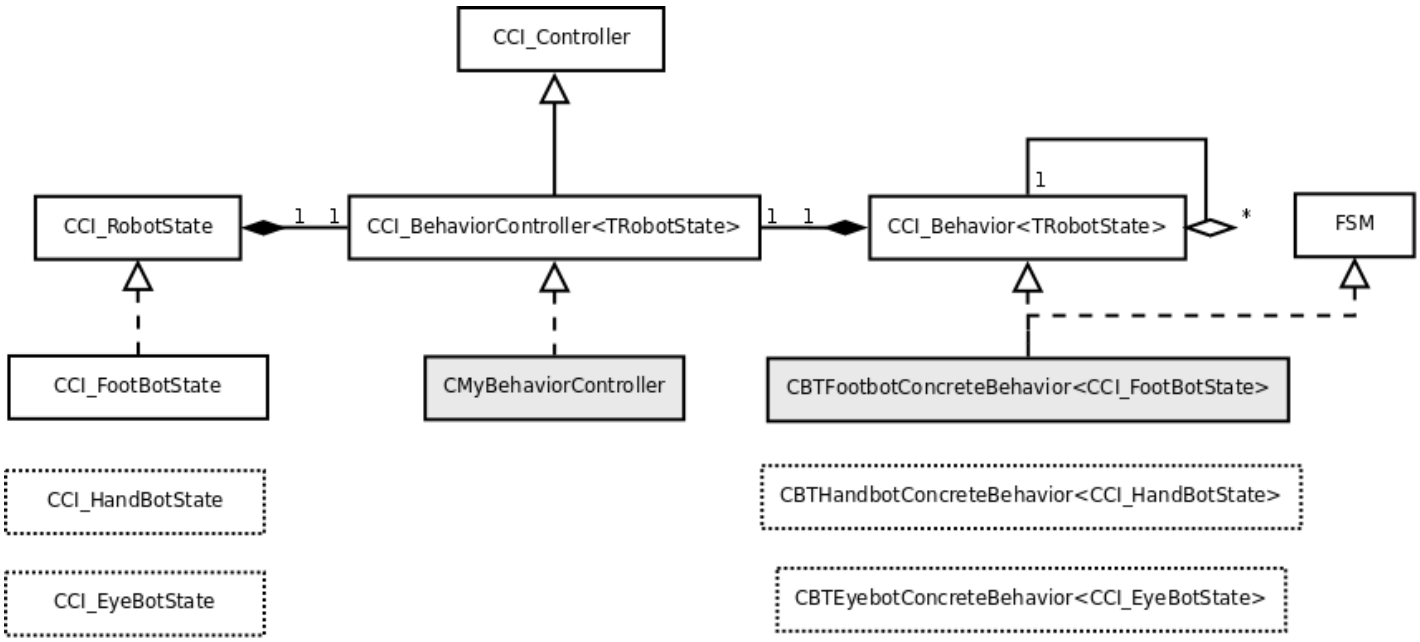
Figure 4.2. The behavioral toolkit with parallel behaviors.



Implementation in ARGoS

Figure 4.3, “Class diagram showing the implementation of the Behavioral Toolkit.” depicts a class diagram of the new behavior control architecture in the simulator. A behavior controller, instead of extending directly from CCI_Controller, extends from CCI_BehaviorController which is a templated class that must be parameterized with one of the three CCI_RobotState classes, according to the robot. The robot state, which is contained inside a behavior controller, is a wrapper around the robot's sensor and actuation space. It's usage will be better depicted in the following two sections. A behavior controller contains also a reference to a CCI_Behavior object, which is the root (main) behavior of the controller. A CCI_Behavior, in turn, can be composed of multiple other behaviors in a arbitrary way. Furthermore, it also a templated class as each behavior will be associated only to a particular robot corresponding to the chosen robot state. Finally, a behavior extends also from the FSM class, which is an utility that allows a more modular development of components (states, transitions) inside a single behavior and a better and standardized inter-communication amongst multiple behaviors. It's usage will be better explained in the section called “Using the FSM”.

Figure 4.3. Class diagram showing the implementation of the Behavioral Toolkit.



How to use it?

Two scripts, ready for your usage, have been created to automate as much as possible the generation of empty skeletons for both behavior controllers and behaviors:

- `$AHSSINSTALLDIR/user/user_scripts/create_behavior_controller.sh`

  To be called from `$AHSSINSTALLDIR/user/<YOUR_USER>/controllers` : This scripts generates a new behavior controller. This means that you will be having a controller class and a separate root behavior class ready. When executed, the script will ask for the (file and class) names for these classes and to associate the controller with a supported (currently only *footbot*, *handbot* and *eyebot*) robot type. Besides, if asked for, this script can also generate a ready-to-go experiment XML associated with the new behavior controller.

- `$AHSSINSTALLDIR/user/user_scripts/create_behavior.sh`

  To be called from `$AHSSINSTALLDIR/user/<YOUR_USER>/controllers/<YOUR_BEHAVIOR_CONTROLLER>` : This script generates a new behavior in case it is called from a directory containing a behavior controller.

Writing a behavior

The following code depicts how to write a simple behavior for a foot-bot. The only method that needs to be overwritten is the CCI_Behavior::Step(CCI_FootBotState& cRobotState). It accepts a CCI_FootBotState object. The robot state is a wrapper around the sensors and the actuators of the robot. It can hence be used to get sensor readings (line 7) and write actuator's output (line 12). All the logic of the behavior can be carried out in between, as described in the commented lines 9-10.

```
void CBTFootbotObstacleAvoidance::Step(CCI_FootBotState& cRobotState) {

  double fLeftSpeed = 0.0;
  double fRightSpeed = 0.0;

  double readings[CCI_FootBotState::NUM_PROXIMITY_SENSORS];
  cRobotState.GetAllProximitySensorReadings(readings);

  // here comes the entire logic which sets
  // fLeftSpeed and fRightSpeed appropriately

  cRobotState.SetFootBotWheelsAngularVelocity(fLeftSpeed, fRightSpeed);
}
```

Depending on what you are aiming at doing, you could even simply combine two or more behavior from the existing behavioral toolkit rather than writing a compeletely new behavior. This list of behaviors, or the behavioral toolkit, is located at `$AHSSINSTALLDIR/user/behavior_toolkit` and is divided into separate (robot specific) folders. Behavior that you write yourself can be combined in exactly the same way, and can be also a contribution to enrich the repository of behaviors.

Writing a combiner behavior (sequentially)

The following code depicts how to write a simple behavior that combine random walk with obstacle avoidance. In the constructor of your behavior, you just instantiate the two sub-behaviors (line 3-4). In the step function, you need to call the the two behavior's step function in the appropriate order. In the example (lines 10-11), random walk's step is called before the obstacle avoidance step. The random walk behavior will be first executed and it will try to propagate it's decision to the actuators. The decision will have effect only if the obstacle avoidance module doesn't detect any obstacles. In case it does, it will overwrite the actuator's variables as appropriate. This *priority* mechanism is ensured by the simple fact that one behavior is called after another. The combination mechanism above assumes that the priority relationship between behaviors is known a-priori. In cases where this is not true, priority can still be guaranteed, with the only constraint that behavior written later will always have an higher or equal priority with respect to the one of behavior's that are already existing and needs to be encapsulated. In this example, the combiner behavior could use the robot state to write to the actuators after lines 10-11, overwriting whatever decision random walk and obstacle avoidance have made.

```
CBTFootbotRWWithOA::CBTFootbotRWWithOA(){
```

```
  m_pcRW = new CBTFootbotRandomWalk();
  m_pcOa = new CBTFootbotObstacleAvoidance(CBTFootbotObstacleAvoidance::VERSION_VECTOR_BASED);

}

void CBTFootbotRWWithOA::Step(CCI_FootBotState& cRobotState) {

  m_pcRW->Step(cRobotState);
  m_pcOa->Step(cRobotState);

}
```

Writing a combiner behavior (parallely)

With the behavioral toolkit there is also the possibility to execute two behaviors in parallel, such that they modify the robot state independently one of each other. The combiner behavior can then combine the two results in the way it sees fit. In order for the two behaviors to not modify the main robot state and to act independently, we created the possibility for the state to be copied with a very easy construct. Lines 11-12 show this process. The two copies are then fed into the two behaviors, and the results are subsequently combined together according to the appropriate logic.

```
CBTFootbotRWWithOA::CBTFootbotRWWithOA(){

  m_pcRW = new CBTFootbotRandomWalk();
  m_pcOa = new CBTFootbotObstacleAvoidance(CBTFootbotObstacleAvoidance::VERSION_VECTOR_BASED);

}

void CBTFootbotRWWithOA::Step(CCI_FootBotState& cRobotState) {

  // Duplicate the state
  CCI_FootBotState pcStateCopy1 = cRobotState;
  CCI_FootBotState pcStateCopy2 = cRobotState;

  m_pcRW->Step(pcStateCopy1);
  m_pcOa->Step(pcStateCopy2);

  // Use the new values in pcStateCopy1 and pcStateCopy2 to code
  // the entire logic here to merge them into cRobotState
}
```

Writing a new behavior controller

The code below shows an example of a behavior controller (i.e. the container). However, such code doesn't need to be written at all, as it is standardized and hence generated automatically by the scripts.

```
int CNewController::Init(const TConfigurationTree t_tree) {

  m_pcState = new CCI_FootBotState(GetRobot());
  m_pcState->Init();
  m_pcRootBehavior = new CBTFootbotRandomWalkWithObstacleAvoidance();
  m_pcRootBehavior->Init();

  return CCI_Controller::RETURN_OK;
}

void CNewController::ControlStep() {
  CCI_BehaviorController<CCI_FootBotState>::ControlStep();
}

void CNewController::Destroy() {

  m_pcRootBehavior->Destroy();

  delete m_pcState;
  delete m_pcRootBehavior;
}
```

Using the FSM

Very often, single behaviors have multiple internal states. The behavioral toolkit provides a standardized way to incorporate these states into a finite state machine (FSM) of a behavior. As the new way of writing behavior controllers introduced in this section is aiming at producing reusable code, the future ARGoS user will have a large repository of ready-to-use behaviors provided to him by other users. This also means that the ARGoS user faces a bigger exposure to (or even base his work on) program code written by others. Hence, the usability of such a modular (blackbox-fashioned) architecture depends upto a certain degree on how easily the internal states of each behavior are accessible and extensible. In an ideal case, such actions should be even possible without having to verify the logical implementation of a behavior in its whole detail, which in turn, contributes to faster development cycles. In the following, examples are shown illustrating the howto implement an FSM in behavior:

Declare all states of the behavior in the header file:

```
class CBTFootBotExampleBehavior: public CCI_Behavior<CCI_FootBotState> , public FSM<string> {

protected:

    // A template class which encapsulates a state of a finite state machine.
    // A state transition is triggered by an input to the state. The datatype
    // of the transition triggering inputs should be provided as the template
    // argument. In this case, it is a string.
    State<string> *m_pcStateOne, *m_pcStateTwo, *m_pcStateThree;

    // Transition condition from state one to two
    string TransitionOneToTwo(CCI_FootBotState& cRobotStateCopy);

    // Transition condition from state two 2 three and vice-versa
```

```
    string TransitionTwo2Three(CCI_FootBotState& cRobotStateCopy);

public:

    // Integral constants for each state (states ID's)
    static const unsigned int STATE_ONE = 0;
    static const unsigned int STATE_TWO = 1;
    static const unsigned int STATE_THREE = 2;
}
```

Initialize the states and declare transitions between the states in the .cpp:

```
CBTFootbotSampleBehavior::CBTFootbotSampleBehavior() :
    CCI_Behavior<CCI_FootBotState> ("demo_bt_behavior") {

    // Initializing the states (parent FSM, state id, string name (for debugging))
    m_pcStateOne = initializeState         (this, STATE_ONE, " [STATE_ONE] ");
    m_pcStateTwo = initializeState         (this, STATE_TWO, " [STATE_TWO] ");
    m_pcStateThree = initializeState     (this, STATE_THREE, " [STATE_THREE] ");

    // Define all the transitions conditions
    m_pcStateOne->addTransition        ("conditionX", m_pcStateTwo);
    m_pcStateTwo->addTransition        ("conditionY1", m_pcStateThree);
    m_pcStateThree->addTransition    ("conditionY2", m_pcStateTwo);

    // Set initial state
    setStartState(m_pcStateOne);

}
```

Input transitions to the states in the control step:

```
void CBTFootbotSampleBehavior::Step(CCI_FootBotState& cRobotState) {

    // Duplicate the state
    CCI_FootBotState cRobotStateCopy = cRobotState;

    if(GetStateID() == STATE_ONE)
    {
        input(TransitionOneToTwo(cRobotStateCopy));
    }
    else if(GetStateID() == STATE_TWO)
    {
        input(TransitionTwo2Three(cRobotStateCopy));
    }
    else if(GetStateID() == STATE_THREE)
    {
        input(TransitionTwo2Three(cRobotStateCopy));
    }

}
```

Implement the transition conditions / methods:

```
string CBTFootbotSampleBehavior::TransitionOneToTwo(CCI_FootBotState& cRobotStateCopy){
    string result = "";

    // if (conditions met to change to STATE_TWO) result = "conditionX";

    return result;
}

string CBTFootbotSampleBehavior::TransitionTwo2Three(CCI_FootBotState& cRobotStateCopy){
    string result = "";

    // if (conditions met to change to STATE_THREE) result = "conditionY1";
    // else if (conditions met to change to STATE_TWO ) result = "conditionY2";

    return result;
}
```

Definition of an Experiment

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

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

```
<ahss-config>
    <!-- ********************************* -->
    <framework>
        ...
    </framework>
    <!-- ********************************* -->
    <controllers>
        ...
    </controllers>
    <!-- ********************************* -->
    <arena size="10,10,3" optimization="2D">
        ...
    </arena>
    <!-- ********************************* -->
    <engines>
        ...
    </engines>
    <!-- ********************************* -->
    <arena_physics>
        ...
```

```
    </arena_physics>
    <!-- ******************************** -->
    <visualisations>
        ...
    </visualisations>
</ahss-config>
```

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

```
<framework>

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

</framework>
```

Tags <clocktick> and <maxclock> together define the total duration of an experiment. In fact, <clocktick> sets the duration of a clock tick in milliseconds[3], while <maxclock> indicates the maximum number of clock ticks[4] after which the experiment is considered finished. The total duration of an experiment in milliseconds is thus given by the following formula:

duration = clocktick x maxclock

Finally, tag <controller_path> contains a colon separated list of directories where controller libraries are compiled.

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

```
<controllers>
    <footbot_sample_controller id="fc" library="footbot_sample_controller">
        <actuators>
            <footbot_wheels>
                <implementation> dummy_footbot_wheels </implementation>
            </footbot_wheels>
            <footbot_gripper>
                <implementation> dummy_footbot_gripper </implementation>
            </footbot_gripper>
        </actuators>
        <sensors>
            <pie_camera>
                <implementation> generic_pie_camera </implementation>
            </pie_camera>
            <footbot_proximity>
                <implementation> generic_footbot_polynomial_proximity
                    </implementation>
            </footbot_proximity>
        </sensors>
        <parameters>
            <min_distance>7</min_distance>
            <wheels_speed>10</wheels_speed>
        </parameters>
    </footbot_sample_controller>
    ...
</controllers>
```

The definition of the objects populating the simulated arena takes place in the section delimited by the tag <arena>. The tag possesses two required attributes: size, which specifies the size of the arena in meters along the main axes of the Swarmanoid Space[5], and optimization, which instructs the arena to store data to favor either 2D physics engines such as the kinematic one, or 3D. When only 2D engines are used, or when the majority of the robots is in 2D engines, the optimization value should be set to 2D, while in the opposite case it should be set to 3D. The rest of the <arena> part is filled with a list of entities. For every entity at least position and orientation are specified; some entities may have more parameters, as shown in the following listing:

```
<arena size="10,10,3" optimization="2D">
    <footbot id="fb">
        <position>9, 9, 0</position>
        <orientation>0, 0, 45</orientation>
        <controller>fc</controller>
    </footbot>
    <eyebot id="eb">
        <position>5, 5, 1</position>
        <orientation>0, 0, 0</orientation>
        <controller>ec</controller>
    </eyebot>
    <block id="obstacle">
        <position>5, 5, .25</position>
        <orientation>0, 0, 0</orientation>
        <size>.5, .05, .5</size>
    </block>
    <cylindric_prey id="cp">
        <position>8, 8, 0</position>
        <orientation>0, 0, 0</orientation>
        <radius>0.10</radius>
        <height>0.10</height>
    </cylindric_prey>
    ...
</arena>
```

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

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

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

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

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

```
<arena_physics>
    <engine id="ground">
        <entity id="obstacle"/>
        <entity id="fb"/>
        <entity id="cp"/>
    </engine>
    <engine id="sky">
        <entity id="eb"/>
    </engine>
</arena_physics>
```

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

```
<visualisations>
    <text_render id="text_world">
        <file>experiment_output.txt</file>
        <precision>2</precision>
    </text_render>
    <opengl_render id="mainwindow">
        <window_size>1024, 768</window_size>
        <window_title>Sample Experiment</window_title>
        <camera_view_XYZ_0>3.30811, 3.36071, 1.9</camera_view_XYZ_0>
        <camera_view_HPR_0>39, -34, 0</camera_view_HPR_0>
        <camera_view_XYZ_1>6.2875, 3.33551, 1.25</camera_view_XYZ_1>
        <camera_view_HPR_1>125, -21.5, 0</camera_view_HPR_1>
        <camera_view_XYZ_2>7.05332, 6.88148, 0.83</camera_view_XYZ_2>
        <camera_view_HPR_2>-155, 0, 0</camera_view_HPR_2>
        <use_textures>true</use_textures>
        <write_frames>false</write_frames>
        <frame_directory>Movies</frame_directory>
        <frame_filename>frame_</frame_filename>
    </opengl_render>
</visualisations>
```

In the above listing, two visualizations are specified: a basic text one, and the 3D OpenGL graphical one. The first accepts as parameters the file name to write the data to and the number of desired decimal numbers. The second, <opengl_render>, sets the window size and title, three camera view points, toggles the use of surface textures to true and video frame storing to false, although a default location and the basename for frames is shown in tags <frame_directory> and <frame_filename>.

A Sample Experiment

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

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

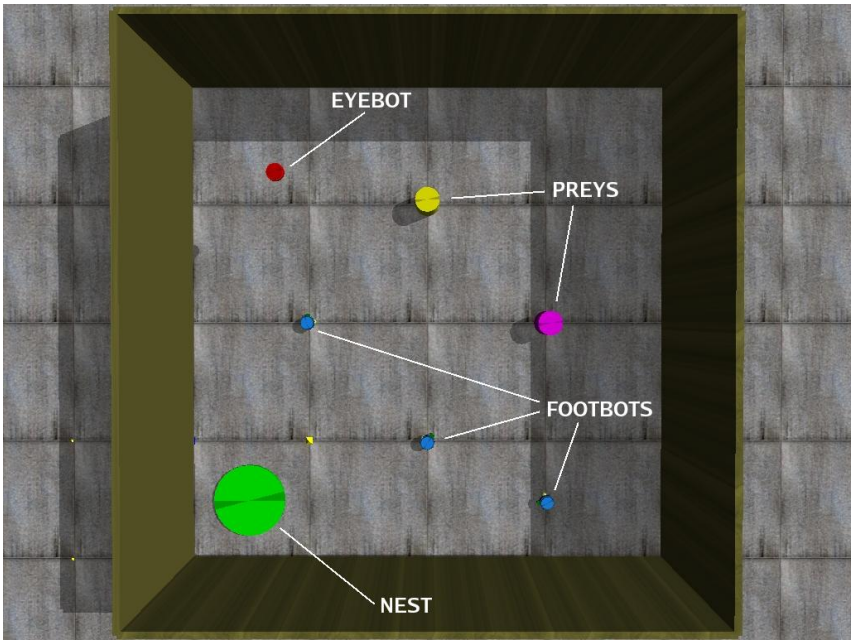
Moreover, to make this experiment more compliant with the control objectives of the Swarmanoid Project illustrated in Chapter ??, we require the robots to cooperate by enriching the task as follows.

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

Furthermore, we want to pursue as much as possible the biological inspiration, avoiding an explicit exchange of structured information with WiFi or Bluetooth. Therefore, communication between the eyebot and the footbots is obtained only by means of visual information. More specifically, footbots inform the eyebot about

their status by means of the colored LEDS they are equipped with, while the eyebot guides the footbots to the prey and to the nest using a laser beam that projects a colored spot on the ground, easily perceivable by the cameras of the footbots.

Figure 4.4. The initial setup of the experimental arena.



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

Figure 4.5. The behavior diagram of the footbot controller.

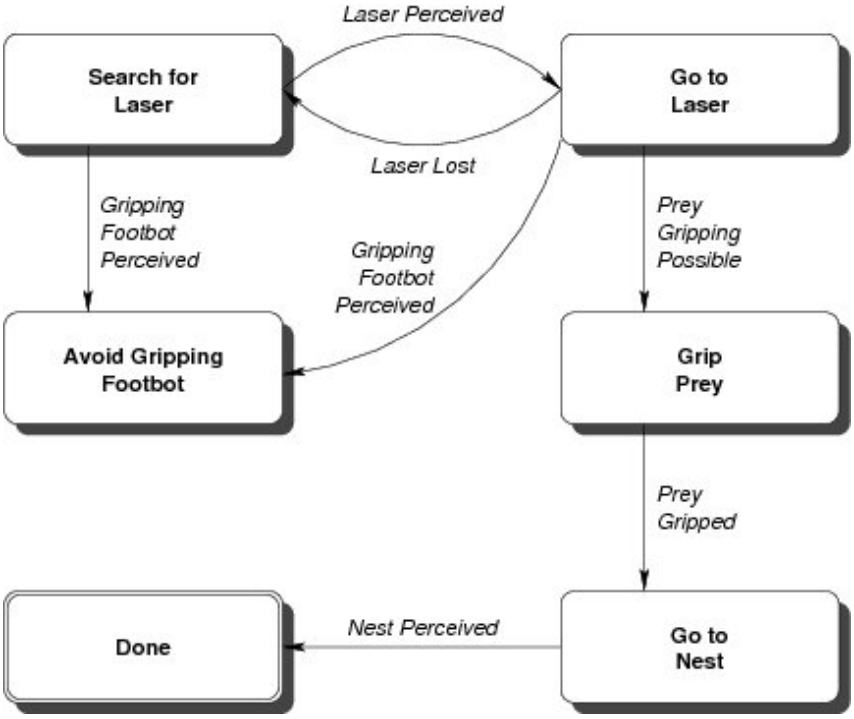
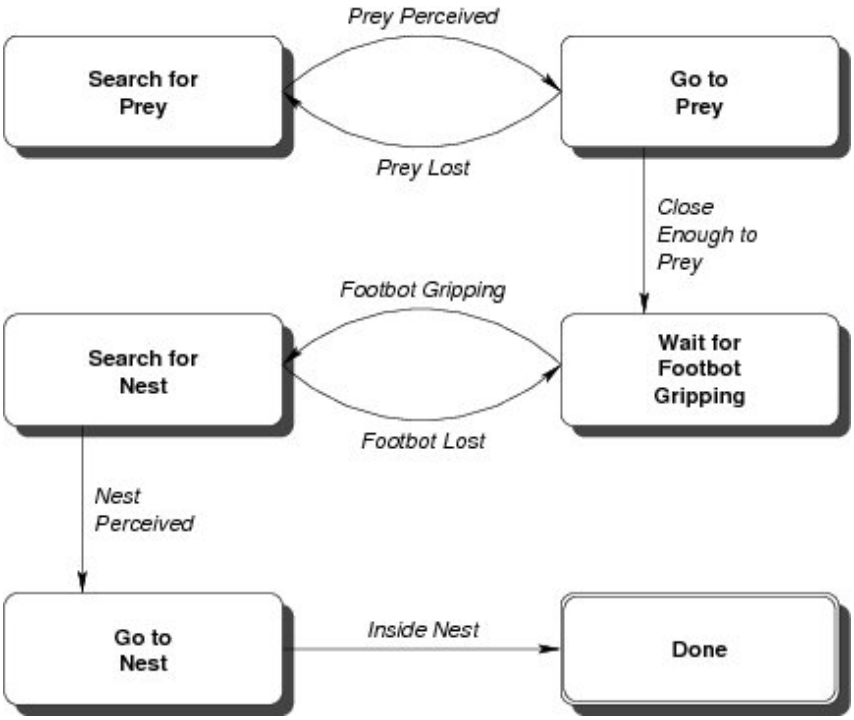


Figure 4.6. The behavior diagram of the eyebot controller.



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

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

When the eyebot perceives the prey, it switches to Go to Prey behavior: the laser is turned on and the eyebot goes towards the prey. When close enough to it, the eyebot stops and waits for the footbots to arrive and grip the prey, switching to Wait for Footbot Gripping behavior.

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

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

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

Figure 4.7. The eyebot waiting for a footbot to grip the prey.

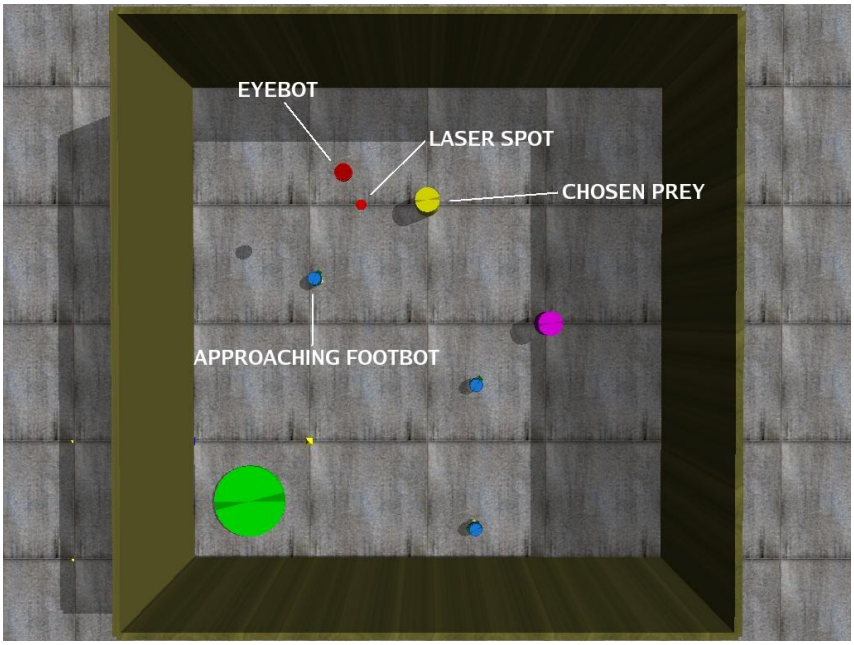


Figure 4.8. The footbot gripping the prey.

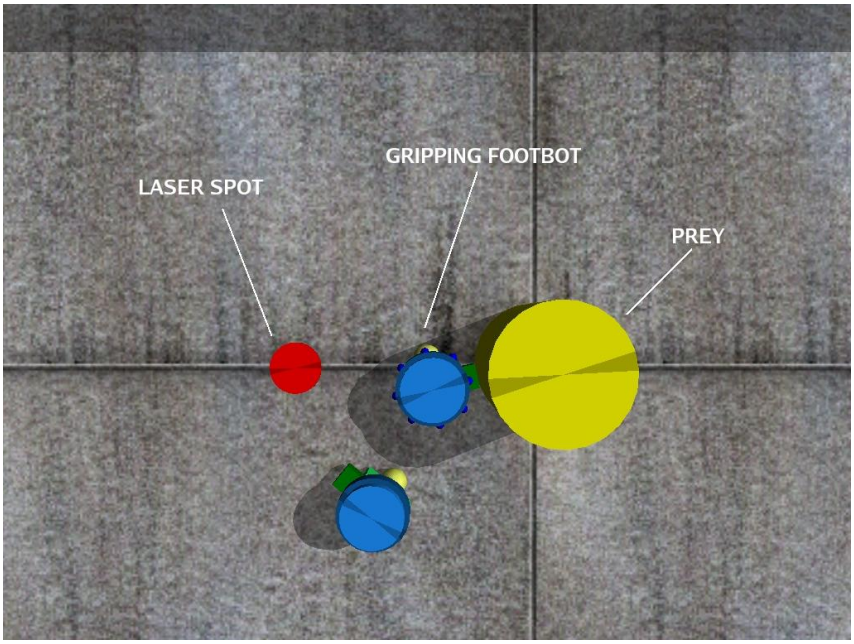


Figure 4.9. The eyebot driving the footbot to the nest.

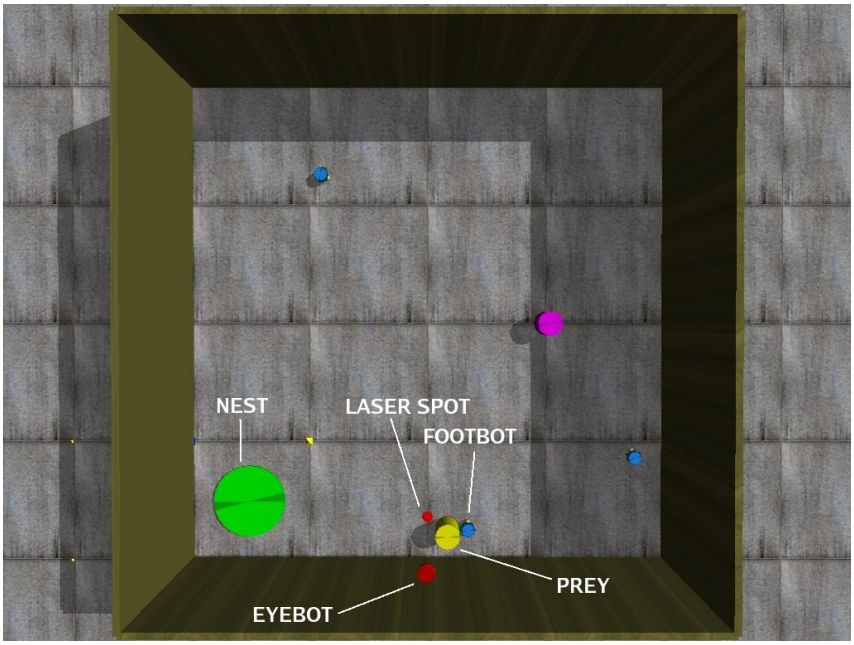
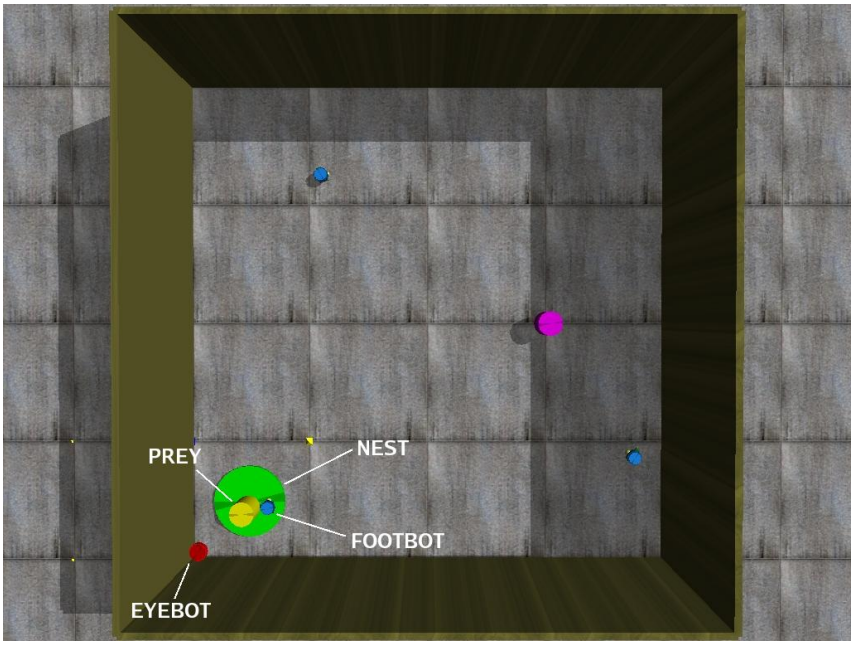


Figure 4.10. The prey is brought to the nest.



[3] At each clock tick the robot control steps are executed.

[4] Therefore, also the maximum number of control decisions.

[5] The arena is a cube.