Definition, Implementation, and Calibration of the Swarmbot3D Simulator

Giovanni C. Pettinaro
giovanni@idsia.ch

Ivo W. Kwee
ivo@idsia.ch

Luca M. Gambardella
luca@idsia.ch

Technical Report No. IDSIA-21-03

December 16, 2003

IDSIA / USI-SUPSI[*]
Instituto Dalle Molle di studi sull' intelligenza artificiale
Galleria 2
CH-6900 Manno, Switzerland

* IDSIA was founded by the Fondazione Dalle Molle per la Qualita della Vita and is affiliated with both the Universita della Svizzera italiana (USI) and the Scuola unversitaria professionale della Svizzera italiana (SUPSI) in Lugano. This research is funded by European Union and the Swiss Government under grant 01.0012

Definition, Implementation, and Calibration of the Swarmbot3D Simulator

Giovanni C. Pettinaro
giovanni@idsia.ch

Ivo W. Kwee
ivo@idsia.ch

Luca M. Gambardella
luca@idsia.ch

December 16, 2003

Abstract

This Technical Report describes the final version of the simulating software Swarmbot3d implementing the swarm-bot simulator as outlined in Workpackage 3, ”Simulator Prototype”.

The document presents all the simulator’s features and acts as a developer’s manual. It describes the implementation choices and software components. It is complemented with the **simulator software** in its final version, and the **Simulator User Manual** for its quick use. The simulator presented here is based on the hardware specifications described in Deliverable 3 (”Simulator Design”) and Deliverable 5 (”User Manual”) of Workpackage 3 of the SWARMBOTS Project.

The development of the simulator Swarmbot3d was intended to address the following supporting roles during this project:

1. To be able to predict accurately both kinematics and dynamics of a single s-bot and swarm-bot in 3D;
2. To evaluate hardware design options for different components;
3. To design swarm-bot experiments in 3D worlds; and
4. To investigate different distributed control algorithms.

Contents

Chapter 1

Introduction

This chapter introduces the reader to the simulator Swarmbot3d and reviews the research context. It starts with a brief taxonomy overview (**Section 1.1**) and a brief survey of close existing software (**Section 1.2**). **Section 1.3** summarizes the main features of Swarmbot3d with respect to the above mentioned taxonomy, and briefly points out both the benefits gained by these specific features and the limitations which they still impose on the environment. Some development notes are briefly synthesized in **Section 1.4**. The last part of this chapter, finally, reports the work which has so far been published on the simulator software herein analysed (**Section 1.5**).

1.1 Simulator Taxonomy

Simulation is a powerful tool for scientists and engineers for the analysis, study, and assessment of real-world processes too complex to be treated and viewed with traditional methods such as spreadsheets or flowcharts ([3]). Indeed, thanks to the ever increasing computational capability of modern computers, simulation has nowadays become an indispensable aid for answering the *what if* type of questions. These have become increasingly common for engineers during the development stage of a new product as well as for scientists during the validation of new working hypotheses of given physical phenomena.

To understand better how simulators can be classified the first step is to consider what they actually are. Basically, a simulation consists in defining an formal description of a real system (model) and in empirically investigating its different behaviours with respect to changing environmental conditions. Since these models could be either static or dynamic in nature, it is plain to draw a first cut here.

Static vs. Dynamic Simulation

Dynamic simulations compute the future effect of a parameter into their model by iteratively evolving the model through time. *Static* simulations, instead, ascertain the future effect of a design choice by means of a single computation of the representing equations. This means that simulators employing static models ignore time-based variances, and therefore they can not for instance be used to determine when something occurs in relation to other incidents. Moreover, because static models hide transient effects, further insights in the behaviour of a system are usually lost.

With the considerable increase of computing power and the consequent decrease of computational cost, the use of simulation has more and more leaned towards the widespread employment of dynamic models. This kind of use has therefore pushed more and more the development of a wide spectrum of simulating tools ranging from specialized applications to general purpose ones. In this respect, two different trends have been shaping: the definition of dedicated simulator languages and the enhancement of those computer programming languages already commonly used. Whatever the case is, a crucial further classification might be drawn with respect to how they let their models evolve.

Continuous, Discrete and Hybrid Simulation

By definition, a dynamic simulation requires its simulated world model to evolve through time. However, the evolution of a simulated system does not necessarily have to be linked directly to the mere passing of time. There are indeed some systems whose evolution could be fully described in terms of the occurrence of a series of discrete events. Because of this, it is possible to distinguish three possible ways of letting time evolve: continuously, discretely, or hybridly between the previous two cases.

1. **Continuous simulation.** The values of a simulation here reflect the state of the modeled system at any particular time, and simulated time advances evenly from one time-step to the next. A typical example is a physical system whose description is dependent on time, such as a water tank being filled with water on one side and releasing water on the other.

2. **Discrete simulation.** Entities described in this sort of simulation change their state as *discrete events* occur and the overall state of the described model changes only when those events do occur. This means that the mere passing of time has no direct effect, and indeed time between subsequent events is seldom uniform. A typical example here is the simulation of a train station where each arrival/departure of a train is an event.

3. **Hybrid simulation.** This type joins some characteristics of the aforementioned two categories. Hybrid simulators can deal with both continuous and discrete events. Such a capability makes them particularly indicated for simulating those systems which have a delay or wait time in a portion of the flow. Such systems might in fact be described either as discrete or continuous events depending on the level of detail required. An example of this type is the simulation of a factory process in which some stages are dependent on the time taken to carry them out.

Swarmbot3d falls in the first category, thus the rest of this document considers just this type of simulators.

Kinematics vs. Dynamics Simulation

At the level of continuous simulation, a further classification refinement can be drawn by examining the type of physics employed.

As mentioned earlier, since continuous simulations evolve their simulated physical world, they need to define mathematical models of it. In this respect, two types of simulators can be distinguished: those using the laws of kinematics only, and those using also the laws of dynamics.

The difference between the two of them is that the former treats every object as a massless pointlike particle and directly modifies its velocity, whereas the latter takes into account both geometry and mass of an object and modifies its velocity only indirectly by applying forces. A simulation involving dynamics is, therefore, more complete however it carries the drawback of a considerable increase in computations.

As clarified later, Swarmbot3d is a physics based simulator, that is it uses the laws of dynamics for evolving its physical systems.

1.2 Survey of Existing Simulator Software

This section surveys some of the most relevant software environments currently available either commercially or as freeware. In this regard, although there exists a great deal of robot simulators, the survey is primarily centred on those that allow simulation of 3D dynamics, that is, those which can handle the physical interaction of multiple robots.

Swarm is a software simulation package for multi-agent simulation of complex systems [6], originally developed at the Santa Fe Institute. Swarm is a *discrete event simulator* (Section 1.1), also referred to as a *dynamic scheduler*. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents. Each agent has a *to-do* list and a reportoire of *actions* that it can perform. The scheduling machinery of Swarm then decides who may take actions and when.

Information on this package is available at its web page:
`www.swarm.org` .

Webots is a commercial 3D simulation package for the Kephera mini-robot as well as for the Koala all-terrain robot family [5]. A recent release has extended its capabilities from pure kinematics simulation to full dynamics ones[1]. It can now also handle any type of robot.

The company selling it provides information at their web page:
`www.cyberbotics.com` .

Darwin2K is a free open-source 3D software package for dynamic simulation and automated design synthesis for robotics. Darwin2K includes a distributed evolutionary algorithm for automated synthesis (and optimization). Its limitation is that currently it only supports a single robot.

The project web page can be found at:
`darwin2k.sourceforge.net` .

Sigel is a development package of control programs for any kind of walking robot architectures using Genetic Programming (GP), a principle which tries to mimic natural evolution combined with the concepts of automatic programming. Its drawback is that it currently supports a single robot only and just GP based learning algorithms.

The software is available at its web address:
`ls11-www.cs.uni-dortmund.de/~sigel` .

Dymola is an object oriented tool for modeling and simulating continuous systems. It is employed mainly in two particular fields: robotics and mechanical systems. The version currently available has been fully integrated with the modelling language Modellica (`www.modellica.org`), which allows the integration and re-use of code developed in different modeling and simulation environments.

The package can be downloaded from:
`www.dynasim.se` .

Dynamechs is a short name for Dynamics of Mechanisms and it is a full multi-body dynamics simulator. The package is a cross-platform object-oriented software developed in order to support dynamics simulation for a large class of articulated mechanisms. Code computing approximate hydrodynamic forces is also available for simulating underwater robotic systems such as submarines, remote operated vehicles (ROVs), autonomous undersea vehicles (AUVs), etc. with one or more robotic manipulators. Joint types supported include the standard revolute and prismatic classes, as well as an efficient implementations (using Euler angles or quaternions) for ball joints.

Work is still ongoing (mainly at the Ohio State University) to extend the capabilities of DynaMechs and develop a user-friendly front-end (called RobotBuilder) which is a useful educational tool for teaching engineering classes both at the undergraduate and graduate level. Further work is also underway for adding the ability to simulate other joint types like the 2 revolute degree of freedom

[1]This is possible thanks to the open source software libraries ODE.

universal (Hooke) joint, and to develop more accurate contact force models to aid in the development of walking machines.

The project provides all the information concerning its current state as well as the software itself at its web page:
`dynamechs.sourceforge.net` .

Havok Game Dynamics SDK is a simulating package built and optimized specifically for video game developers. It is currently one of the most powerful real time physical simulation solution commercially available. It offers an integrated rigid body, soft body, cloth, and rope dynamics with one of the fastest collision detection algorithms and one of the most robust constraints across all consumer level hardware platforms.

All the information concerning its features and the licenses costs can be found at the company's web page:
`www.havok.com` .

Mirtich and Kuffner's Multibody Dynamics is a software package based on Dr. Mirtich doctoral research. Its main characteristic is that its libraries compute the forward dynamics of articulated tree-like structures of rigid links by using Featherstone's algorithm. The user specifies the inertial properties of each link, as well as the connectivity between links. In this version of the package, however, only prismatic, revolute, and floating joint types are supported.

Further information can be found at:
`www.kuffner.org/james/software` .

MuRoS is a simulating environment fully developed at the University of Pennsylvania [2]. It allows the simulation in 2D of several multi-robot applications such as cooperative manipulation, formation control, foraging, etc.. Overall it is a good package but its restriction to a bi-dimensional world greatly limits its predictions to simple environments.

Details on the project can be found at:
`http://www.grasp.upenn.edu/~chaimo/muros_download.html` .

ODE is a free, industrial quality library for simulating articulated rigid body dynamics, such as ground vehicles, legged creatures, and moving objects in VR environments. It is fast, flexible, robust and platform independent. Among other characteristics it possesses advanced joints, contact with friction, and built-in collision detection.

Differently from similar packages, its stability is not sensitive to the size of each time step. Moreover, it supports real hard constraints and it has no difficulties in simulating articulated systems. This is due to the fact that the package is neither a purely Lagrange Multiplier based technique, which facilitates the flexible management of hard constraints at the expense of the simulating performance for large articulated systems, nor is it a purely Fetherstone based method, which, despite its high simulating speed, does only allow tree structured systems and does not support hard contact constraints.

The software and further information about it can be found at the project's web page:
`www.q12.org/ode` .

Vortex is a commercial physics engine for real-time visualization and simulation currently released at its 2.1 version. Providing a set of libraries for robust rigid-body dynamics and collision detection, Vortex is unsurpassed for any 3D interactive simulation requiring stable and accurate physics.

Among the various characteristics of the software in its current state is a new more stable solver delivering dynamics with rotational effects generated by the shape and mass of the simulated objects. Worth noticing, it is also the availability of a new scalable friction model with better accuracy for stacked objects and for situations that require a closer approximation to Coulomb friction. Another important point is the significant speed improvements in its latest version, which allows developers to simulate larger numbers of objects and more complex environments.

The software does not impose any restrictions on the number of simulated objects: the size of a simulation is basically dependent just on the speed of the CPU and the amount of memory available.

The company provides all the information concerning the specific features of the software and the costs of its license at their web page:
`www.cm-labs.com`.

1.3 Characteristics of Swarmbot3d

Using the taxonomy overview discussed in Section 1.1 and the research and technological context shown in Section 1.2, it is now possible to introduce the main features of Swarmbot3d:

- **Continuous 3D dynamics.** Swarmbot3d is a continuous 3D dynamics simulator of a multi-agent system (swarm-bot) of cooperating robots (s-bots). The simulator has at its core the physics solver provided by Vortex™.
- **Hardware s-bot compatibility.** All hardware functionalities of the s-bot have been implemented in Swarmbot3d. The simulator is capable of simulating different sensor devices such as IR proximity sensors, an omni-directional camera, an inclinometer, sound, and light sensors.
- **Software s-bot compatibility.** Programs that are developed using Swarmbot3d can be ported directly to the hardware s-bot due to a common application programming interface.
- **Interactive control.** Swarmbot3d provides online interactive control during simulation useful for rapid prototyping of new control algorithms. Users can try, debug and inspect simulation objects while the simulation is running.
- **Multi-level models.** Most robot simulation modules have implementations of different levels of detail. Four s-bot reference models with increasing level of detail have been constructed. **Dynamic model switching** is an included feature that allows switching between representation models in real-time. This allows the user to switch between a coarse and detailed level of simulation model to improve simulation performance in any moment.
- **Swarm handling.** The capability of handling a group of robots either as independent units or in a swarm configuration, that is an entity made of s-bots which have established connection with other peers. These connections are created dynamically during a simulation and can deactivated when the components disband. Connections may also be of rigid nature giving to the outcoming structure the solidity of a whole entity. This feature is unique with respect to other existing robot simulators.

1.4 Development Notes

Swarmbot3d is a complex project. The simulator consist of more than 10,000 lines of C++ code, covering more than 50 classes. Besides that, it uses the Vortex™SDK (for the physics), OpenGL (for the viewer), C/C++ (core coding), Python (for the embedded interpreter), XML (file format).

Concerning the main programming tools, we have used Swig (to generate Python wrappers), PHP (to generate some parts of the XML files), CVS (for the code repository), umbrello and medoosa (for UML diagrams), Latex and Doc++ (for the documentation).

The developments have been done on Intel based workstations (1.8 GHz, dual Pentium) with hardware accelerated graphics card (NVidia Quadro Pro) using Linux (Debian 3.0).

1.5 Presentation of Published Work on Swarmbot3d

The development work carried out on the simulator software herein analysed has been initially introduced in two previous project deliverables (Deliverable 3, ”Simulator Design” and Deliverable 5, ”Simulator Prototype” of Workpackage 3).

A first public presentation of the whole software environment has been carried out in the 33rd International Symposium on Robotics of the International Federation of Robotics [9]. This work briefly shows how, with the aid of Swarmbot3d, it is possible to design service applications of swarm robotics.

A second work carried out on the simulator and published in Autonomous Robots shows how the detailed model of one s-bot compares with its real counterpart [7]. This work also shows how the various s-bot abstractions compare with the detailed one.

A final work lately submitted to the IEEE International Conference on Robotics and Automation (ICRA) shows the application of a simulation technique specifically developed in Swarmbot3d. Such a technique allows to reduce the simulation time by employing always the robot model which best suits the terrain environment onto which it has to move [8].

Chapter 2

Modeling and Design

This chapter examines the design details characterizing the simulation environment Swarmbot3d. The chapter starts by presenting an outline of the designing principles onto which the development of the simulator has been based (**Section 2.1**). The next section presents the different simulation models of the s-bot that have been created (**Section 2.2**) and gives a detailed description of each of the functional modules of the simulated robot. **Section 2.3** presents the architecture implementing an artificial swarm. Finally, **Section 2.4** describes the modeling of the simulated environment adopted by the simulator.

2.1 Simulator Design

Swarmbot3d has been designed having three main features in mind:

1. modular design
2. multi-level modeling, and
3. dynamic model exchange.

As clarified later, such a multi-featured design philosophy allows to build a flexible and efficient simulator.

2.1.1 Modular design

Modularity was deemed to be a necessary characteristic because it allows users to have a large freedom in customizing their own swarm according to their specific research goals. This revealed to be very useful during the early prototype stages of the robot hardware when its specifications often changed.

Since it was necessary to keep the simulation models parallel to the hardware developments of the prototype, the use of modularity allowed not only to remodel a specific s-bot geometry but also to extend Swarmbot3d with new mechanical parts or new sensor devices which became available throughout the hardware development.

This section will describe the different s-bot simulation modules that have been implemented in Swarmbot3d.

2.1.2 Multi-level modeling

This was deemed an important characteristic too, because, by providing different approximation models for the same part, an end-user is given the possibility to load the most efficient and functionally equivalent abstraction model among those available to represent the real s-bot.

Following this line of reasoning, one s-bot may be loaded for instance as a detailed model for an accurate simulation or it may be loaded as a crude abstraction for big swarm evaluation, where the accuracy of a single robot is not a crucial research element. People working with learning techniques or emerging intelligence might in fact be more interested in simple coarse s-bot models. Conversely, those experimenting with the interaction within a relatively small group of units (between 5 and 10) might instead be more keen on using a more refined s-bot model.

Viewed in this sense, the possibility of having different levels of abstraction greatly extends the types of users who might employ Swarmbot3d as a useful research tool. The advantage provided by the availability of a hierarchical abstraction for a robot is twofold: it frees users from the programming details of creating an appropriate model for the robot and it allows them to concentrate their efforts on a much higher research level.

Next chapter (Chapter 3) expands this discussion by introducing 4 reference models (Fast, Simple, Medium and Detailed) which allow to represent one s-bot at different levels of detail.

2.1.3 Dynamic model exchange

Using the hierarchical abstraction levels as introduced above, a way to change the s-bot representation during simulation has been implemented. The availability of such a feature allows, for example, a user to start a simulation with the most simple abstraction level for one s-bot when the terrain onto which it moves is flat and switch to a more refined model representation when the environment or interaction among s-bots require a more detailed treatment. Dynamic model changing allows Swarmbot3d to be as fast as possible by introducing complexity only when it is needed.

A requisite, though, is that all representation models need to be *compatible* with each other, *i.e.*, all models need to show a similar behaviour when a particular robot command is issued. It is therefore important to adjust each model so that speed, mass, and geometry are calibrated to ensure compatibility with each other, even if they differ in representation detail.

Chapter 4 presents some validation experiments which show the compatibility among the aforementioned reference models.

2.2 S-Bot Modeling

An artificial swarm of robots (swarm-bot) may be viewed as a very complex entity made of a large collection of units (s-bots). One s-bot, if viewed in all its details, is quite a complex system, and simulating it at that level would very easily lead to a very high level of complexity without necessarily adding any particular insights.

To avoid this problem, the first step is to identify the most important characteristics for the different types of users envisioned, and then extract them into a data structure. By doing so, it is possible to build a complete description of the state in which one s-bot may find itself in and at the same time to allow users to customize opportunely an entire swarm.

The robot prototype's functionalities have been thoroughly studied and the relevant functional modules have been identified. Table 2.1 lists the breakdown of one simulated s-bot into its functional modules.

As reported later (Chapter 3), 4 simulation **reference models** have been chosen to represent one s-bot at increasing levels of detail. These models are made out of an opportune composition of various modules with increasing levels of detail. Since this chapter will indicate which module implementation belongs to which s-bot model, the reference models are shortly listed below.

- **Fast:** a very simple s-bot abstraction. It is a miniature robot in an artificially low-gravity simulation environment which allows fast computations. It has a rotating turret and sample based sound and infrared sensors.

Table 2.1: Breakdown of one s-bot simulation model by functional module. The right column indicates the type of the module.

module	**type**
Treels Module	actuator
Turret Module	actuator
Front Gripper Module	actuator
Flexible Side Arm	actuator
Camera Module	sensor
Proximity Sensor Module	sensor
Ground IR Sensor Module	sensor
Sound Module	actuator/sensor
Light Ring Module	actuator/sensor
Inclinometer Module	sensor
Temperature and Humidity Sensor	sensor
Battery Voltage Meter	sensor

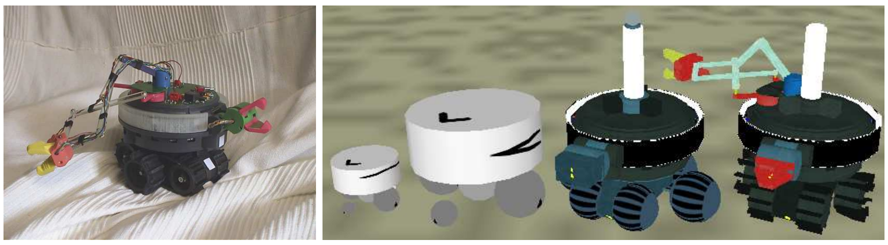


Figure 2.1: Real s-bot (left) and the four simulation models representing the real s-bot (right).

- **Simple:** a properly scaled-up version of the previous model roughly approximating hardware dimensions in a simulated environment with real gravity pull.
- **Medium:** a more detailed version of the Simple model using a 6-wheel locomotion system and a rotating turret featuring all supported sensors (sound, light, infrared, camera).
- **Detailed:** the most detailed version of the simulated s-bot with teethed wheels and flexible side-arm gripper. This model replicates in details the geometrical blue prints of the real hardware (Figure 3.4) as well as masses, centre of masses, torques, acceleration, and speeds (cf. Deliverable 4, ”Hardware Design”).

Figure 2.1 shows a prototype of the hardware s-bot and the 4 simulation models. Clearly, the more refined and detailed the model of each s-bot is, the more CPU demanding a simulation gets as the number of robots increases. This particular topic is clarified later on in this document. A more thorough description of these models is deferred to Chapter 3.

Modularity has been enforced by designing the software using strict object oriented programming principles. Each functional module is implemented by its own class (in the `simulation/modules/` sub-folder). For modules which have more than one implementation (*e.g.*, a simple and a more detailed approximation), a base class hierarchy has been defined to ensure compatibility among the different implementations. Most modules are composed of several sensor and actuator devices, such as torque sensors, light sensors, servo motors or LED’s. These components are implemented as separate devices in the sub-folder `simulation/devices/`.

The remaining of this section describes each module in detail according to the following points:

- **Description:** it reports shortly the functionality of the module;
- **Modeling:** it describes how this part is modeled in the simulator;
- **Comparison with hardware:** it compares and describes the differences of the simulated module with its hardware counterpart;
- **Simulation parameters:** it lists the main parameters used in the module;
- **State variables:** it provides a synopsis of the functionality of the module in terms of its abstract state;
- **Class description:** it provides a short description of the program class implementing the module;
- **User programming interface:** it lists the user programming functions available in the module.

2.2.1 Treels Module

Description

One s-bot employs as its own means of locomotion a tracks system typical of the track-laying vehicles. The rubber caterpillar tracks provide good traction in rough terrain situations, while the slightly larger central wheels enable good on-the-spot turning capability. This peculiar system has been patented by the LSI lab at the EPFL with the name of *Treels*©.

Modeling

It is clear that the simulation of a detailed geometrical representation of the real treels system is too cumbersome. The solution adopted to approximate the wheels with the simulator is to define it by a set of 6 wheels evenly distributed on each robot side with the middle wheels slightly outward and larger than the others. Depending on the type of terrain used, 3 different treels approximating models have been implemented (the labels in bold refer to the reference s-bot model that employs that particular implementation):

1. **Fast/Simple:** Only the two central wheels are simulated. This means that this treels abstraction is reduced to a 2-wheeled robot. However, two passive caster wheels are placed in front and on the back of the treels body to provide balancing support.
2. **Medium:** The treels are in this case approximated by 6 spherical wheels. This abstraction has the central wheels modeled slightly larger than the remaining 4 wheels, thus it is geometrically closer to the real s-bot.
3. **Detailed:** The treels system is in this case modeled very accurately: the model is defined with 6 teethed cylindrical wheels which replicate those of the real s-bot. The teeth provide the additional traction and grip not available with the spherical wheels.

The Treels model can be viewed in Figure 2.2 where the three approximation models for this subsystem are shown.

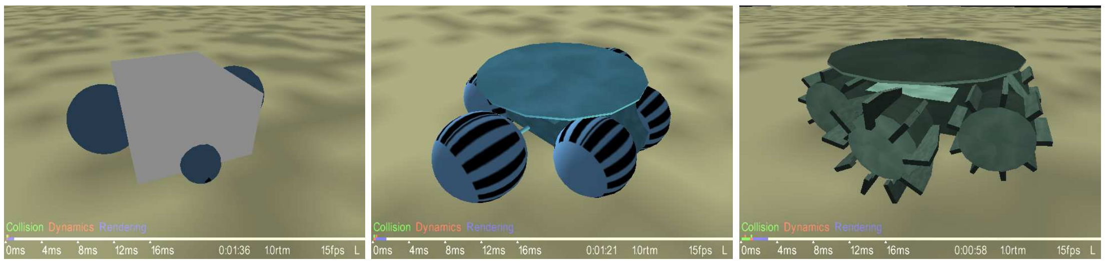

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:01:36 10rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:01:21 10rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:58 10rtm 15fps L

Figure 2.2: Simulation models of the s-bot's tracks system (treels) in isometric view. From left to right: Fast/Simple, Medium, and Detailed models.

Comparison with hardware

It has been experimentally observed for the detailed treels model with 6 teethed wheels that, in situations where a sharp edge falls just perpendicular between the front and central wheels, the simulation model looses grip. This would not occur with the real robot, since the presence of a rubber band joining the inner wheels would still provide contact with the surface. Except this specific case, however, this model abstraction using 6-teethed wheels seems to approximate sufficiently well the macro-behaviour of the real treels system in almost all situations.

The reference position of the treels is defined at the geometric center of the the treels main body. The center-of-mass is defined in relation to this reference point. The real s-bot has its center of mass placed slightly forward from the center mainly due to the asymmetry of the placement of the front gripper motors. Such an asymmetry causes one s-bot to ”lean forward” in its stable position. Except for the Fast and Simple models, all other abstractions implement this shift of the center-of-mass.

Simulation parameters

Table 2.2 summarizes the main simulation parameters of the Treels module. These parameters are based on hardware data as provided in ”Hardware Analysis” of Milestone 5. The table also indicates where these parameters are defined: some of the parameters are solely defined in the XML file, whereas others are defined in the code.

State variables

Taking into account the description of the Treels simulation module, the *state* of the track system can be identified by the following tuple (Figure 2.3)

$$\langle \vec{x}, \vec{p}, v_l, v_r \rangle \tag{2.1}$$

where

$\vec{x}$	is the treels reference position,
$\vec{p}$	is a unit vector pointing in the heading direction, and
v_l, v_r	are the velocities of left and right tracks, respectively.

Notice that these state variables are not meant to be directly manipulated by the user. The physics engine keeps track of them and updates their values at each time step. The application user interface (API) and the `TreelsModule` class provide methods for accessing these variables.

Table 2.2: Simulation parameters of simulated treels module.

Parameter	**Value**	**Defined in**
treels body type	sphere (fast/simple)	*_treels.me
	detailed geometry (medium/detailed)	*_treels.me
joint type	hinge	*_treels.me
number of wheels	2 (fast/simple)	*_treels.me
	6 (medium/detailed)	*_treels.me
center wheel radius	2.2 cm	*_treels.me
front/back wheel radius	2.0 cm	*_treels.me
back-front axle distance	8.0 cm	*_treels.me
front wheels separation	4.2 cm	*_treels.me
center wheels separation	8.2 cm	*_treels.me
COM[1]: (x,y,z)	(0,-0.5,0) cm	*_treels.me
MOI[2]: (m_{xx}, m_{yy}, m_{zz})	(338.5, 1943, 1771)	*_treels.me
mass	250 g	*_treels.me
gear ratio	150:1	-
torque unit[3]	24412 dyne·cm (=2.4412 mNm)	*_treels.me
maximum torque	$2.1971 \cdot 10^6$ dyne·cm (=90 units)	*_treels.me
speed unit	0.28 cm/s	TreelsModule.h
maximum speed	20.0 cm/s (=70 units)	TreelsModule.h
friction[4]	$12 \cdot 10^4$ dyne	World_Settings.me

1) Center of mass relative to body origin
2) Moment of inertia matrix.
3) Measured at wheel axle, after 150:1 gear reduction.
4) Isotropic friction due to limitations in physics engine.

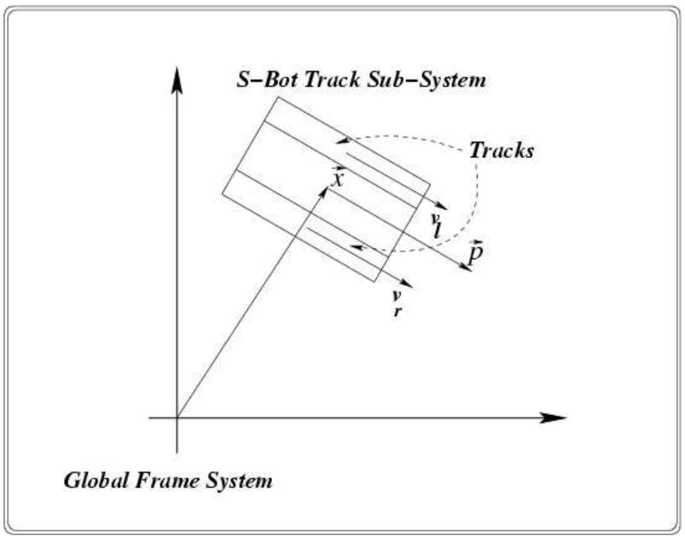


Figure 2.3: Track system and global frame system.

Class description

The treels module is implemented by the `TreelsModule` class. Because `TreelsModule` is derived from the class `VortexObject`, it inherits all methods from this base class. The behaviour of the treels can be monitored or modified by directly accessing the class methods available. For example, one can get the current position, or get current direction by using the commands:

```
void TreelsModule::getPosition(double pos[3]);
void TreelsModule::getDirection(double dir[3], int axis);
```

or to set the wheels passive one can use

```
void TreelsModule::setPassive();
```

A UML class diagram for the TreelsModule and related modules are given in Appendix B.

User programming interface

Users should access the treels module using the high-level user interface as defined in the standard s-bot API (see Appendix A). The API is implemented by the class `VirtualSBot`. Below we describe the API functions for the treels module.

`void setSpeed(int left, int right);`

This command sets the speed of the robot tracks. It has the same effect both on the real hardware and on the simulated s-bot. It takes two arguments:

`left` Left motor speed.

`right` Right motor speed.

According to the various combination of values `(vleft, vright)`, it is possible to perform straight motion, rotation about the central axis, and smooth route along a curved path. Notice that because of limited motor torque, the requested speed is never achieved immediately. Furthermore, this is just a requested speed, thus if for some reason the wheels get blocked externally, then the requested speed will not be achieved at all.

Both `left` and `right` can be any integer values within the range $[-127;127]$. Notice, however, that because of physical limits of the motor torques, the maximum speed a real s-bot can reach is no more than 200 mm/s. Since a speed unit correspond to 2.8 mm/s, the maximum speed reachable is equivalent to about 70 units. With the real hardware the robot speed is software limited at 90 units, although users should never use speeds higher than 70. Because of this speed is software limited within Swarmbot3d at 70 units.

`void getSpeed(int *left, int *right);`

This command reads the current speed of each track. It takes two arguments:

`left` Left motor speed pointer.

`right` Right motor speed pointer.

The values returned by this reading are within the range of $[-127;127]$.

`void getTrackTorque(int *left, int *right);`

This command reads the current torque of each track. It takes two arguments:

`left` Left track torque pointer.

`right` Right track torque pointer.

The values returned by this reading are within the range of $[-127;127]$. See Table 2.2 for units.

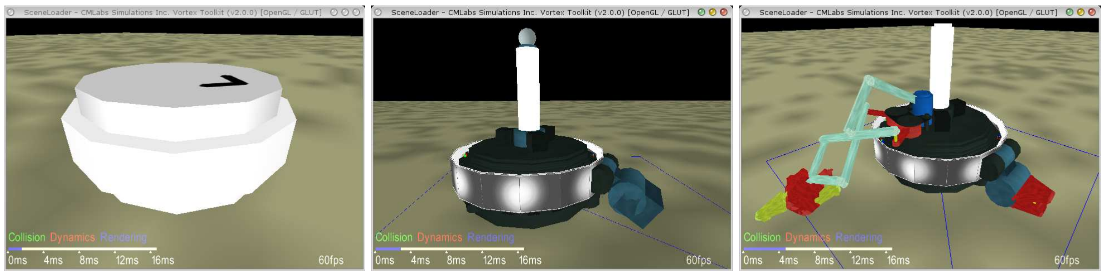

SceneLoader - CMLabs Simulations Inc. Vortex Toolkit (v2.0.0) [OpenGL / GLUT]
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms
60fps

Figure 2.4: 3D Simulation model of s-bot's turret. From left to right: Fast/Simple, Medium, and Detailed models.

2.2.2 Turret Module

Description

The Turret of one s-bot is a cylindrically shaped part rotating about a central axis on top of the underlying treels system, which has been discussed in the previous section. The turret of the real s-bot houses all the electronic boards and it includes most sensor modules, such as the infrared proximity sensors, light sensors, sound sensors etc. A semi-transparent ring surrounding the turret can act as a gripping target for other s-bots using either the front or the side-arm gripper.

Modeling

The rotating Turret has 3 implementations available in Swarmbot3d, which differ only in their level of geometric detail and in the type of sensors they can support (the labels in bold refer to the reference s-bot model employing that implementation):

1. **Fast/Simple:** The model for the Simple s-bot has a single cylinder with roughly the height and diameter of the real turret. The model for the Fast s-bot is simply a half sized version of it. In both cases, just a sample based look-up table for infrared sensors and an effort sensor are available.

2. **Medium** This approximation is similar to the detailed model. It replicates the turret in very fine detail but it does not support a side arm. It provides all sensors available on the real robot (sound, light, LED ring, infrared, force/torque sensors, effort sensor, omni-directional camera).

3. **Detailed** This model adds to the real turret replica presented for the Medium model a side arm. It incorporates, besides the sensors listed above, the IR optical barrier on the side arm gripper jaws.

Figure 2.4 shows the three abstraction models for the s-bot Turret.

In terms of dynamics, all turret simulation models have the same complexity, *i.e.*, a single dynamic body fixed to the treels using a motorized hinge. This means that as far as the dynamics computation is concerned, there is no efficiency gain (or loss) among the various turret models presented above. However, the differences in efficiency among them become evident when their geometric and functional description is considered. These differences are due to the collision handling, 3D rendering, and sensor simulation implied by their use.

Comparison with hardware

No major differences have been observed with the mechanical behaviour.

Table 2.3: Simulation parameters of turret module.

Parameter	**Value**	**Defined in**
joint type	hinge	s-bot_*.me
number of bodies	1	*_turret.me
max. turret diameter	5.7 cm	*_turret.me
turret height	4.6 cm	*_turret.me
COM[1]: (x,y,z)	$(4, 1.1, 0)$ cm	*_turret.me
MOI[2]: (m_{xx}, m_{yy}, m_{zz})	(1625, 3136, 1625) g·cm^2	*_turret.me
rotation unit	2°	TurretModule.h
rotation speed	0.5 rad/s	TurretModule.h
rotation range	[-200,200] units	TurretModule.h
gear ratio	720:1	-
torque unit[3]	117176 dyne.cm (=11.7176 mNm)	TurretModule.h

1) Center-of-mass with respect to body reference.
2) Moment of inertia.
3) Measured at wheel axle, after 150:1 gear reduction.

Simulation parameters

The main parameters of the turret module are summarized in Table 2.3. The moment of inertia (MOI) has been estimated by approximating the complex geometry of the turret with a cylindrical shape, thus some errors with the ”real” MOI may be expected. The center of mass (COM) is measured with respect to the geometric center of the main cylinder and has been estimated roughly from the hardware prototype within about 10 mm precision; it is slightly forward located due to extra weight of the front gripper motor.

Class description

Programming commands for the turret are available using directly the methods included in the `TurretModule` class. Since `TurretModule` is derived from `VortexObject`, all methods from this base class are available for the `TurretModule`, too. Additionally, the `TurretModule` provides locking and unlocking of the rotation servo motor. The servo motor is implemented in the class `ServoModule` and provides low-level actions for controlling the motorized hinge.

A UML class diagram for the TurretModule and related modules are given in Appendix B.

User programming interface

The standard s-bot API as implemented by the class `VirtualSBot`, provides high-level commands for controlling the Turret. These functions are described here below:

```
void setTurretRotation(int pos);
```

The command takes one argument only:

`pos` Angle of rotation.

This command drives the upper part (Turret) of one s-bot for the specified angle about the robot central vertical axis. The maximum Turret rotation which a real s-bot is able to carry out clockwise or counter-clockwise corresponds to 200°. Since units are expressed in units within the range of $[-100; 100]$, each unit is equivalent to 2°.

The invocation of this command instructs a low-level controller thread (called by the class `ServoModule`) to rotate the simulated turret to a desired position indicated by `pos`. Notice that, as it happens with the real s-bot, it may take some time to reach the desired position because of the finite rotation speed.

`int getTurretRotation(void);`

This command reads the angle about which the s-bot turret has rotated with respect to the default zero rotation. This particular position corresponds to the robot turret aligned with the underlying treels and with its front gripper right on top of ground sensor 0.

It does not take any argument but it returns the turret rotation angle in units within the range of $[-100;100]$.

`int getTurretRotationTorque(void);`

This command reads the rotation torque of the robot turret. It does not take any argument and it returns an integer within the range of $[-127;127]$ expressing the torque read. Refer to Table 2.3 for the units.

2.2.3 Front Gripper Module

Description

The front gripper enables one s-bot to grasp an object or another fellow s-bot, in which case s-bots build a physical connection. Connections can be rigid enough to allow one robot to lift the gripped s-bot.

Modeling

The simulated Front Gripper has been implemented in three increasing levels of detail:

1. **Fast/Simple:** The Fast and Simple s-bot models do not explicitly incorporate a Front Gripper model but connection between two s-bots can still be achieved by creating a static joint in the XML file description. Note that this approximation is *static*, i.e. new connections cannot be made during simulation. This is the preferred connection model for the Fast and Simple reference models.

2. **Medium:** The Front Gripper is approximated in this case by a ”sticking element”, a rectangular box which is able to attach to other s-bots in a ”magnet-like” fashion. The box is hinged on the turret and can be elevated up and lowered down. This extra degree of freedom is important for coordinated motion tasks on rough terrain. Since this implementation model is defined as a separate geometric model connected to the Turret with a hinge, it introduces one extra dynamical body. Note that this approximation is *dynamic*, i.e. connections can be turned on/off during simulation. It is intended to be used for the Medium reference model.

3. **Detailed** This Front Gripper model approximates the hardware gripper in very fine detail. It is characterized by

   - a hinge for elevating or lowering the gripper arm,
   - two jaw elements (upper and lower) with which to realize the gripping (opening and closing),
   - integrated optical light barrier.

   This gripper implementation introduces 3 extra dynamical bodies: one for the arm body and two for the jaws. This approximation is *dynamic* by nature, i.e. connections can be created and cancelled by opening and closing the jaws. It is intended to be used for the Detailed reference model.

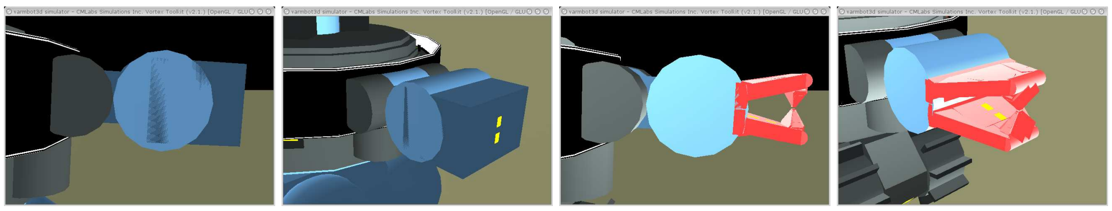


Figure 2.5: Simulated models of Front Gripper. Left to right: Medium Front Gripper side and ISO view; Detailed Front Gripper side and ISO view.

Table 2.4: Simulation parameters of front gripper module.

parameter	value	defined in
joint type	hinge	*_turret.me
number of bodies	1 (medium)	Front_Arm/Coarse
”	3 (detailed)	Front_Arm/Detailed
grasping force at jaw	15 N	Front_Arm/Detailed/front-arm.me
arm elevation torque	0.5 Nm	*_turret.me
zero set point	15°	-
jaw element speed	1 rad/s	GripperModule.cpp
rotation speed	0.87 rad/s	GripperModule.cpp
light barrier max sensitivity	40 mm	GripperModule.cpp
light barrier min sensitivity	1 mm	GripperModule.cpp

Figure 2.5 shows the three simulation models for the s-bot Front Gripper.

Using any of the gripper simulation models, a user can create connections between s-bots. during simulation. The sizes of the different gripper approximations have been adjusted, so that the distance between two connected s-bots using any of the gripper models are the same.

Comparison with hardware

The detailed version of the Front Gripper has been calibrated with the elevation speed of the arm and the opening/closing speed of the jaws. In this respect, the simulation model shows a similar “time-delay” between issuing, for example, the gripper command “close” and the actual time that the gripper has “closed”. This “time-delay” is not present in the Simple and Medium representations as the sticking element sets its state immediately.

Simulation parameters

The simulation parameters for the gripper models are summarized in Table 2.4.

State variables

Abstracting from the particular model selected, the simulation *state* of the s-bot gripper can be characterized by the following (implicit) state variables:

$$\langle elevation, aperture, grippedBody \rangle \tag{2.2}$$

where

elevation	the elevation angle of the gripper arm
aperture	the opening angle of jaw elements
grippedBody	the body ID of the connected object

Class description

The Simple, Medium and Detailed Front Gripper models are implemented as derived classes (`GripperModuleSticky` , `GripperModuleWithArm` and `GripperModuleWithJaws`) of the common base class `GripperModule` and provide extra methods that are not available through the standard s-bot API (Appendix A). Developers who need to extend or modify the default simulated gripper functionalities need to access these classes directly.

Notice that the ID of the connected object in the state variable *grippedBody* (Eq. 2.2) is not available in the real robot: it is only present in simulation. For this reason, the standard API does not allow access to *grippedBody* and its value can only be accessed through the class `GripperModule`. The `MdtBodyID` of the gripped body (state variable *grippedBody* in Eq. 2.2) may be retrieved using the command

`GripperModule*::getGrippedBody()`.

The connection between one robot's front gripper and the other robot's light ring attachment is in the hardware s-bot purely mechanical. Ideally the *Detailed* Gripper model using the jaws should replicate this behaviour but, because of the large forces involved in simulation, such a tight grip can not be reproduced. An alternative implementation realizing a rigid grip uses *dynamic joints*. This technique allows to create a rigid joint between the gripper and the other s-bot every time a rigid connection between two s-bots is requested. The dynamic connection of the sticky elements (in the simple and medium gripper models) works exactly in this way.

Sticky elements (with `MATERIAL_ID=5`) can only stick to "stick-able" objects (with `MATERIAL_ID=4`) and these are determined by setting the appropriate material ID in the model section of the XML file. A connection, *i.e.*, the creation of a dynamic joint, can only occur when a "sticking-element" is in contact with a "stick-able" object and at the same time a closing command is issued to the gripper (`setGripperAperture(0)`). If the element is not in contact with a "stick-able" object, a dynamic joint is not created. The sticky element is implemented in the class `StickyElement`.

A UML class diagram for the GripperModule and related modules are given in Appendix B.

User programming interface

End-users should control the gripper using gripper commands as defined by the standard s-bot interface through the class `VirtualSBot`. All gripper implementations accept these commands but return a void action if a command is irrelevant. The Simple and Medium Gripper models overload the jaw commands with equivalent commands to control their sticky elements.

```
void setGripperElevation(int elevation);
```

This command sets the elevation of the robot Front Gripper Arm expressed in sexagesimal degrees.

It takes one argument:

`elevation` Angle of elevation in degrees.

Because of physical constraints, the real gripper arm elevation is limited within the range of $[-90;90]$. Within Swarmbot3d, any value outside such a range is automatically cut to the nearest end. This means, for instance, that requesting an elevation of 100° would result in stopping the arm actuation as soon as 90° is reached.

The actual action is performed by a low-level background thread (in the class `ServoModule`) that implements the control of the elevation actuator. Notice that it takes a finite time to reach the desired elevation position. If the gripper arm is obstructed in some way, and the elevation position cannot be reached, the thread is killed after a timeout period. For the Minimal Gripper, elevation actions result in a void action and any return value is undefined.

```
int getGripperElevation(void);
```

This command reads the current gripper arm elevation with respect to the zero elevation which occurs when the arm and the Turret are aligned. It does not take any argument but it returns an integer within the range of $[-90;90]$.

```
void setGripperAperture(int aperture);
```

This command sets the aperture of the Front Gripper Arm jaws. The value specified corresponds to the sexagesimal angle between one jaw and the Front Arm central axis.

It takes one argument:

`aperture` Angle of aperture in degrees.

The maximum angle a real jaw can reach is $45°$. This means that the maximum bite width reachable is $90°$. Hence, aperture can be any integer within the range of $[0;45]$.

As shortly mentioned above, the aperture command also controls the ”sticking-elements” of the Medium Gripper model. Although this last does not have jaw elements, setting the aperture to any non-zero value results in ”unsticking” any previous connection, and setting `aperture=0` results in creating a dynamic joint if the element is in contact with some stick-able object.

```
int getGripperAperture(void);
```

This command reads the current elevation of one Front Gripper Arm jaw, which doubled becomes the overall aperture of the gripper. It does not take any argument and it returns an integer within the range of $[0;45]$.

```
int getGripperElevationTorque(void);
```

This command reads the gripper torque. It does not take any argument and it returns a value within the range of $[-127;127]$. It may be used, for example, to estimate the weight of gripped objects.

```
void getGripperOpticalBarrier(unsigned *internal,
                              unsigned *external,
                              unsigned *combined);
```

This command reads the optical barrier of the front gripper arm. The readings are within the range of $[0;700]$ with 0 representing total obstruction and 700 representing open empty gripper.

It does not take any input parameter, but it returns three values:

`internal` This is a pointer to the reading representing the internal barrier sensor.

`external` This is a pointer to the reading representing the external barrier sensor.

`combined` This is a pointer to the reading representing the internal and external combined barrier sensor reading.

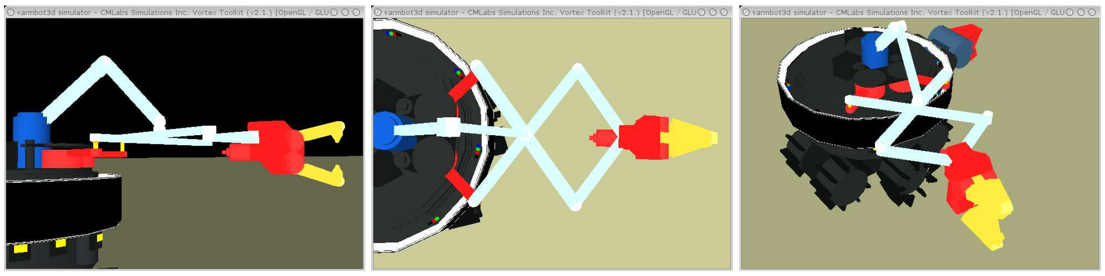


Figure 2.6: The Side Arm is modeled as a 3D extendible/retractable scissor-like mechanism. From left to right: side, top and isometric view.

2.2.4 Flexible Side Arm Module

Description

The hardware design specifications (Workpackage 2, Milestone 3) describe the Flexible Side Arm as a 3D extendible/retractable scissor-like mechanism actuating three inner joints. An extra gripping device is also located at the utmost tip of this structure. The connection to other s-bot occurs by closing this extra gripper, whereas, conversely, a disconnection occurs by opening it.

Compared to the Front Gripper Arm, the Side Arm is intended for a different purpose. While the former is rigid and powerful enough to lift another s-bot, the latter is flexible and it allows links extended over bigger distances. The Flexible Arm is not rigid nor strong enough to lift or push objects and it is mainly intended to be used, when fully extended, either to pull objects or to provide fellow s-bots with lateral support for creating stable connected swarm structures. Also the gripping force is reduced to 1 N (it is 15 N for the rigid Front Gripper Arm).

Modeling

The only simulation model of the Side Arm is a *fully detailed* one which replicates all elements of the scissor-like structure including a detailed jawed gripper element (Figure 2.6). This is mechanically speaking the most complex part of the simulated s-bot: featuring 11 dynamical bodies and using 12 joints. The implementation of the Flexible Arm with this level of detail introduces a fair amount of computational overhead in the simulator and can greatly affect the efficiency of the overall simulator. The use of the Flexible Arm simulation module should therefore only be included for experiments using the Flexible Side Arm explicitly.

As far as control is concerned, the Side Arm module simulates 5 servo motors: three independent servo-motors for controlling the extension of the arm, and two controlling the jaws.

Simulation parameters

The simulation parameters for the Side Arm are summarized in Table 2.5. The elevation torque of the side-arm (0.04 Nm) is much less compared to that of the rigid front gripper (0.5 Nm). Also the gripping force is reduced to 1 N from the 15 N of the rigid gripper.

State variables

The *state* of a side arm might be synthesized with the following tuple

$$\langle \phi_s, \gamma_s, d_s, \psi_s \rangle, \tag{2.3}$$

Table 2.5: Simulation parameters of flexible Side Arm module.

parameter	**value**	**defined in**
number of motors	5	side_arm.me
number of joints	12	side_arm.me
number of bodies	11	Side_Arm/
horizontal torques	0.07 Nm	side_arm.me
vertical torque	0.04 Nm	side_arm.me
unit	0.458°	SideArmModule.cpp
grasping force at jaw	1 N	side-arm-gripper.me

where ϕ_s, γ_s are the elevation and rotation of the Side Arm structure relative to its rest position, d_s is the current depth of the Side Arm, and ψ_s is the current aperture angle of the jaws.

Class description

The Side Arm is implemented in the class `SideArmModule` that contains 5 simulated servo-motors, three for controlling the arm position and two synchronized simulated servos that control the gripper jaws. The servos are accessible as members of the class `ServoModule` which provides low-level threaded actions for controlling the motorized hinges to rotate to a specific angular position (similar to the hardware PIC controllers).

A UML class diagram for the SideArmModule and related modules are given in Appendix B.

User programming interface

The state of the simulated Side Arm can be controlled and read out using the standard API commands (Appendix A):

```
void setSideArmPos(int elevation, int rotation, int depth);
```

This command positions the Flexible Side Arm in space by setting the elevation and rotation angle of the main Side Arm axis, and by setting the outward/inward depth it has to protrude/retract, respectively.

It takes three input parameters:

`elevation` This represents the tilting angle in sexagesimal degrees with respect to the horizontal Flexible Arm resting position.

`rotation` This represents the rotating angle in sexagesimal degrees with respect to the Flexible Arm resting position which is 110° from the Front Gripper Arm main axis.

`depth` This is the depth to which the gripper at the end tip of the arm has to be protruded or retracted.

Notice that the command does not directly modify the state variables as described in Eq. 2.3 but use the three servo motors to position the Side Arm. Inverse kinematic control is needed to map servo-motor positions *elevation*, *rotation* and *depth* to the corresponding state variables.

```
void getSideArmPos(int *elevation, int *rotation, int *depth);
```

This command reads the current position of the Flexible Side Arm.

It does not take any input parameters but it returns three values:

`elevation` Tilting angle of the arm main axis with respect to the horizontal rest position.

`rotation` Rotating angle of the main axis with respect to the resting position (110° space from the Front Gripper Arm main axis).

`depth` Protruding or retracting depth to which the gripper of the Flexible Arm has to be located.

```
void getSideArmTorque(int *elevation, int *rotation, int *depth);
```

This command reads the current torque to which the Flexible Arm motors are subjected.

It takes no input parameters and it returns three values:

`elevation` This is a pointer to the torque reading of the elevating actuator.

`rotation` This is a pointer to the average torque reading of the actuators performing horizontal rotations.

`depth` This is a pointer to the difference between the torque of the actuators performing horizontal actuation.

```
void setSideArmOpened(void);
```

This command actuates the Side Arm Gripper so as to open up completely its jaws. It does not take any input parameters nor it returns any value.

As opposed to the Front Gripper, the user API does not allow to control the aperture of the Side Gripper by specifying an angle: in case it is necessary, the user should refer to the the implementation class.

```
void setSideArmClosed(void);
```

This command actuates the Side Arm Gripper so as to close completely its jaws. It does not take any parameters, nor returns anything.

```
void getSideArmOpticalBarrier(unsigned *internal,
                              unsigned *external,
                              unsigned *combined);
```

This command reads the optical barrier of the Flexible Gripper Arm. The readings are within the range of $[0;700]$ with 0 representing total obstruction and 700 representing open empty gripper.

It does not take any input parameters and it returns three values:

`internal` This is a pointer to the reading representing the internal optical barrier sensor reading.

`external` This is a pointer to the reading representing external optical barrier sensor reading.

`combined` This is a pointer to the reading representing the internal and external combined optical barrier sensor reading.

2.2.5 Camera Module

Description

The real s-bot has an omni-directional camera mounted at the top of the Turret which provides a fish-eye view of the surrounding environment. The camera looks at a spherical mirror suspended above it at the top of a vertical transparent tube. The camera is mainly intended to monitor the surrounding environment and to detect the colors of the light-ring of other s-bots.

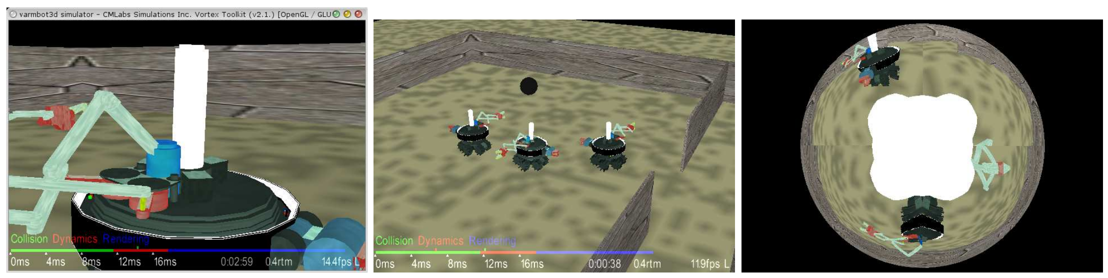

varmbot3d simulator - CMLabs Simulations Inc. Vortex Toolkit (v2.1.) [OpenGL / GLU]
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms

Figure 2.7: From left to right: (a) The simulated Camera is place on top of the center pole. (b). ISO-view of scene. (c). Simulated fish-eye camera view seen from center s-bot.

Modeling

Two simulation models are available for the camera.

1. **Fast/Simple**: A high-level Camera implementation (`FakeCamera`) which simulates methods for high-level recognition of observable objects. The fake camera senses all LED's that can be seen in a certain scope maximum distance and vertical sensing angle. It is placed on top of the s-bot and fixed to the body with a vertical offset. It is a 360° camera. The LEDs are placed around the s-bots body radius and they can have different color.

   This implementation skips resource-intensive low-level image processing computation and may be much more efficient if only high-level cognition function of object is required. It is intended to be used for the Fast and Simple reference models.

2. **Medium/Detailed**: A low-level Camera implementation (`RawCamera`) which returns an array of values representing color values of a CCD camera (as in the real s-bot). This camera has been implemented in the simulator by placing the OpenGL camera at the position of the spherical mirror at the top of the tube. Because the OpenGL libraries do not provide a command to generate an omni-directional view directly, 4 snapshots are taken, each covering a 90° field of view. The 4 snapshots are stitched and mapped to polar coordinates to simulate a fish-eye view. It is intended to be used for the Medium and Detailed reference models.

The cameras are instantiated by the constructor of the `robot/VirtualSBot` class. Figure 2.7 shows the camera fixture (the center pole structure), an arbitrary scene, and the omni-directional camera view as seen from the central s-bot.

Comparison with hardware

The stitching procedure of the `RawCamera` is not perfect and leaves some artifacts at the stitched boundaries. These artifacts are known to be more pronounced at inclined postures of the s-bot.

Simulation parameters

The simulation parameters for the two simulated types of Camera are summarized in Table 2.6 and 2.7.

State variables

The simulation *state* describes the current camera view and comprises the list of viewable objects for the `FakeCamera`, or the entire video buffer for the `RawCamera`.

Table 2.6: Simulation parameters of raw camera module.

parameter	value	defi ned in
vertical offset	11.85 cm	*_turret.me
buffer size	640 × 448	(window size)

Table 2.7: Simulation parameters of the fake camera module.

parameter	value	defi ned in
vertical offset	5.0 cm	FakeCamera.h
maximum sensor distance	1000 cm	FakeCamera.h
sensing vertical angle	70°	FakeCamera.h

Class description

This raw camera module is only available by default for the *medium* and *detailed* s-bot models, and is implemented in the class `devices/RawCamera`. The fake camera is implemented in the class `devices/FakeCamera` and can be used on all reference s-bot models.

A UML class diagram for the CameraModule and related modules are given in Appendix B.

User programming interface

Currently, there is no camera command defined in the standard API yet and the use of the camera must be done using the class implementations directly.

The high-level `FakeCamera` implementation will probably not be compatible with the standard s-bot API and should not be used when portability to the real s-bot is in mind.

For the `RawCamera`, the current camera view can be read by accessing the contents of the camera buffer through a pointer to an array of type `char` (8 bits). The size of the array is $640 \times 448 \times 3$, repeating the three RGB values for each pixel of the camera pixel array.

2.2.6 Proximity Sensor Module

Description

The s-bot has 15 infra-red (IR) Proximity Sensors that can detect obstacles in the immediate neighbourhood of one s-bot. The sensors are placed at the lower part of the Turret with angular distance of 1/18 of a full circle, but 3 sensors in front of the s-bot (where the Front Gripper is placed) are ”missing”. This is important, because one s-bot cannot sense its vicinity using the IR sensors along the direction of the Front Gripper.

Infrared sensors detect the amount of reflected infrared light emitted by the robot IR LED. The sensitivity of such sensors on the real s-bot has a sensitivity volume which can be approximated by a cone with an angle of about 40° and a range of about 200 mm.

Modeling

The simulator provides two implementations of the infrared proximity sensors.

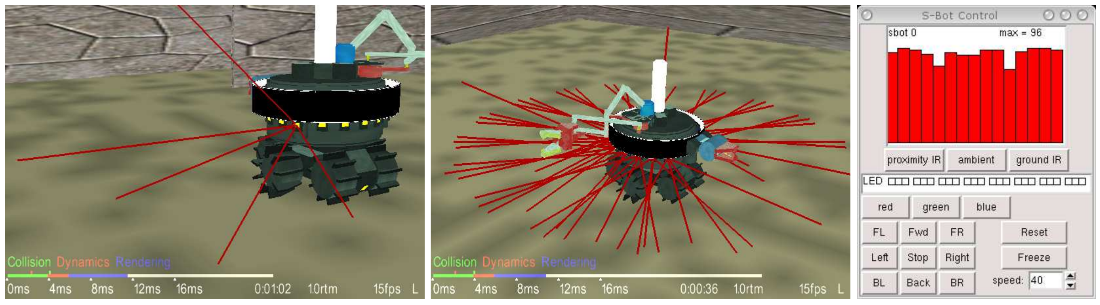

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:01:02 10rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:36 10rtm 15fps L
S-Bot Control
sbot 0
max = 96
proximity IR
ambient
ground IR
LED
red
green
blue
FL
Fwd
FR
Reset
Left
Stop
Right
Freeze
BL
Back
BR
speed: 40

Figure 2.8: From left to right: (a) A single IR sensor is modeled using 5 probing rays. (b) ISO-view of s-bot using Proximity Sensors. (c) GUI frame showing Proximity Sensor values.

1. **Fast/Simple:** A sample-based implementation using a look-up-table (LUT) is available. This class implements sampling techniques applied to generic sensors which depends only on the relative positions of s-bots (no walls, obstacles, or shadowing is taken into account in this first implementation). The LUT values need to be measured separately performing robotic experiments. Because currently only 2D LUT values are available, this type of proximity sensor is mainly meant for simulations employing the Fast and Simple reference models on flat plane.

2. **Medium/Detailed:** The ray-tracing based Proximity Sensor simulation module models each infrared sensor using 5 rays that are distributed in the cone sensitivity volume: 1 central ray, and 4 peripheral rays on a cone surface (Figure 2.8). Proximity is detected by monitoring the intersection of each line with other bodies (*e.g.* other s-bot or walls). This type of proximity sensor can be used independently from terrain type, and has been integrated in the Medium and Detailed reference models.

Because of its general validity in complex situations, the *ray-tracing* based sensing model is preferred above the *sample-based* implementation in all but simple environments. For these last, the sample based approach may be more efficient. The rest of this section discusses only the ray-tracing based proximity sensor model.

Comparison with hardware

In the ray-tracing based implementation, for each sensor, 5 rays detect any intersection of nearby objects and compute an average distance value which is then converted to an integer sensor response. Such a response has been calibrated using linear regression (in a log-log space) using measured distance responses of 3 different IR sensors using a total of 33 measured values.

Based on the experimental data, a linear fit in log-log space appeared sufficient to provide an adequate fit using the regression function:

$$y = \exp(a + b * \log(x)) \tag{2.4}$$

where y is the Proximity Sensor response, x is the distance value, and a and b are the regression parameters. The best fit was found using a nonlinear least squares Marquardt-Levenberg algorithm. The final set of regression parameters and their asymptotic standard errors found were:

$$\begin{aligned} a = &\quad 8.44352 && \pm 0.2134\ (2.528\%) \\ b = &\ -0.777545 && \pm 0.05392\ (6.935\%). \end{aligned} \tag{2.5}$$

One difference between the simulated IR sensors and the real ones is that while the latter detect any object within a sensitivity cone, the former suffer from the so-called ”dead spot”, *i.e.*, the space in which

Table 2.8: Simulation parameters of the Proximity Sensor module.

Parameter	**Value**	**Defined in**
number of sensors	15	modules/ProximitySensorModule
minimum sensing distance	0.2 cm	modules/ProximitySensorModule
maximum sensing distance	20 cm	modules/ProximitySensorModule
ray-tracing cone angle	30°	devices/IRSensor
ray-tracing per sensor	5	devices/IRSensor
noise level	uniform 10 %	devices/IRSensor

small objects remain undetected because they lie just in between rays. Although the sensitivity field of the IR sensor was measured to be about 40°, it was decided to place the peripheral ray-tracing rays at 30° angles around the center ray. The dead spot is in this way reduced. Indeed, it could be reduced even further by using more probing rays at an increase of computational costs.

Another difference which has been observed is that real sensors are sensitive to the color of the reflecting surface, whereas the simulated ones (solely based on intersection distance) are not.

Uniform noise of 10% has been added to the distance measurements in order to simulate sensor noise. Additionally, the hardware IR sensors seem to suffer from noise due to cross-talk with the power supply but this is expected to get improved in future designs. Thus, such a contribution to the noise in the simulator has not been modeled.

Simulation parameters

The simulation parameters for the simulated IR Proximity Sensors are summarized in Table 2.8.

State variables

The simulation *state* of the Proximity Sensor describes the current proximity sensor values and can be represented by

$$\langle\, sensor[15]\,\rangle, \tag{2.6}$$

where $sensor[i]$ is the sensor reading of the i-th proximity IR sensor.

Class description

The IR proximity module is implemented by the class `ProximitySensorModule`. The sensor objects are implemented by the class `devices/IRSensor`. The position of the sensor is determined by the `IR_SENSOR_xx` tags in the `*_turret.me` XML file. The x-axis of its model geometry determines the normal (sensing) direction of the sensor.

A UML class diagram for the `ProximitySensorModule` is given in Appendix B.

User programming interface

The standard s-bot interface provides commands to read the value of the Proximity Sensors. To read a single IR sensor use the command

```
unsigned getProximitySensor(unsigned sensor);
```

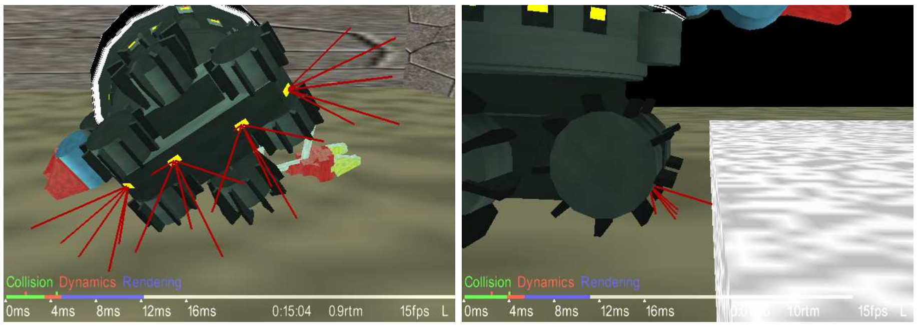

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:15:04 0.9rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 1.0rtm 15fps L

Figure 2.9: Left: Capsized robot showing 4 Ground Sensors at the bottom of the Treels. Right: S-Bot using Front Ground Sensor to detect a step.

This command reads the current value of the specified IR Proximity Sensor. It takes one input parameter and it returns one unsigned integer encoding the distance to the hit target within the range of $[0;8191]$:

`sensor` IR labelling number within the range of $[0;14]$.

The distance encoded is not linearly proportional to the actual distance. One object 150 mm away produces a reading of 20, another one at 50 mm away induces a reading of 50, yet another one at 10 mm produces a reading of 1000, and finally one at 5 mm away produces a reading of 3700.

```
void getAllProximitySensors(unsigned sensors[15]);
```

This command reads the value of all IR Proximity Sensors and dumps them into an array. All readings are within the range of $[0;8191]$. It does not take input parameters but it returns its readings in an array of 15 locations:

`sensors` Array of distances to objects.

```
void setProximitySamplingRate(SamplingRate rate);
```

This command sets the IR Proximity Sensor sampling rate but for the simulator this command is irrelevant because in the simulator the IR values are calculated in real time. It takes one input parameter:

`rate` Sampling rate expressed in terms of SAMPLING_RATE_xx values.

2.2.7 Ground IR Sensor Module

Description

Four IR sensors are placed at the bottom of the Treels body looking downward to the ground. These Ground Sensors can be used to detect terrain conditions such as holes or edges. The left image in Figure 2.9 shows cap-sized s-bot showing its 4 ground sensors at the bottom. The right image in Figure 2.9 shows the use of the front ground sensor to detect a step during navigation.

Table 2.9: Simulation parameters of the Ground Sensor module.

Parameter	**Value**	**Defined in**
number of sensors	4	modules/TreelsModule
minimum sensing distance	0.1 cm	modules/TreelsModule
maximum sensing distance	6 cm	modules/TreelsModule
ray-tracing cone angle	30°	devices/IRSensor
number of rays per sensor	5	devices/IRSensor
noise level	uniform 10 %	devices/IRSensor

Modeling

Similarly to the IR Proximity Sensors, the Ground Sensors response is modeled by linear regression of experimentally measured sensor values versus distance in log-log space. However, as opposed to the case of lateral IR previously discussed, ground sensors show a much shorter maximum sensitivity range (20 to 30 mm). A linear fit in log-log space was assumed using the function:

$$y = \exp(a + b * \log(x)) \tag{2.7}$$

where y is the Proximity Sensor response, x is the distance value, and a and b are the regression parameters. The best fit was found using a nonlinear least squares Marquardt-Levenberg algorithm. The final set of regression parameters and their asymptotic standard errors found were:

$$\begin{aligned} a = &\quad 7.59789 &\pm 0.7081\ (9.32\%) \\ b = &\quad -2.12704 &\pm 0.2537\ (11.93\%). \end{aligned} \tag{2.8}$$

Comparison with hardware

The simulated Ground Sensors have the same limitations as the lateral IR sensors, *i.e.*, they cannot distinguish colors and they suffer of the ”dead spot” syndrome, however this last is not so crucial in this case thanks to the much more limited sensing range.

Simulation parameters

The simulation parameters for the simulated IR ground sensor are summarized in Table 2.2.7.

State variables

The simulation *state* of the Ground Sensor describes the current IR sensor values and can be represented by

$$\langle\ sensor[4]\ \rangle, \tag{2.9}$$

where $sensor[i]$ is the sensor reading of the i-th ground IR sensor.

Class description

The Medium and Detailed Treels models support the use of simulated Ground IR Sensors. The sensor objects are implemented by the class `devices/IRSensor`, the same class that is used for the simulated IR Proximity Sensors. The position of the sensor is determined by the `IR_SENSOR_xx` tags in the `*_treels.me` XML file. The x-axis of its model geometry determines the normal (sensing) direction of the sensor.

User programming interface

The standard s-bot API (Appendix A) defines three commands with regard to ground sensors:

```
unsigned getGroundSensor(unsigned sensor);
```

This command reads the value of one of the specified Ground Sensor. It takes one input parameter and it returns one unsigned integer within the range of $[0;255]$:

`sensor` Ground IR Sensor number within the range of $[0;3]$.

The distance to the ground encoded in the reading of one ground IR sensor is not linearly proportional to the actual distance. A ground 6 mm away provides a reading of 1, one at 2 mm produces a reading of 25, and one at 1 mm distance generates a reading of 100.

```
void getAllGroundSensors(unsigned sensors[4]);
```

This command reads the value of all Ground Sensors and dumps them into an array of 4 elements. It takes no input parameters and it returns the Ground Sensor readings into the specified array:

`sensors` Array of distances to objects.

```
void setGroundSamplingRate(SamplingRate rate);
```

This command sets the sampling rate of the IR Ground Sensors. However for the simulator this command is irrelevant because the IR values are calculated within the simulator in real time. It takes one input parameter and it does not return anything:

`rate` Sampling rate expressed in terms of `SAMPLING_RATE_xx` values.

2.2.8 Light Ring Module

Description

Around one s-bot's Turret, there is a semi-translucent ”light ring” which serves multiple purposes.

1. It can act as a gripper attachment for the Front and Side Arm grippers of other robots so as to create fixed or flexible swarm-bot structures,

2. It features 8 colored light groups that can display patterns of color. Each group can be controlled independently to blink and its color can be specified independently using RGB values. The light ring is primarily intended for communication between other s-bots that can recognize the colors using the omni-directional camera.

3. Each sector also includes a Light Sensor that can measure the intensity of (green) ambient light. Using these Light Sensors, one s-bot can measure the direction and intensity of the light in its environment, for example for navigation tasks.

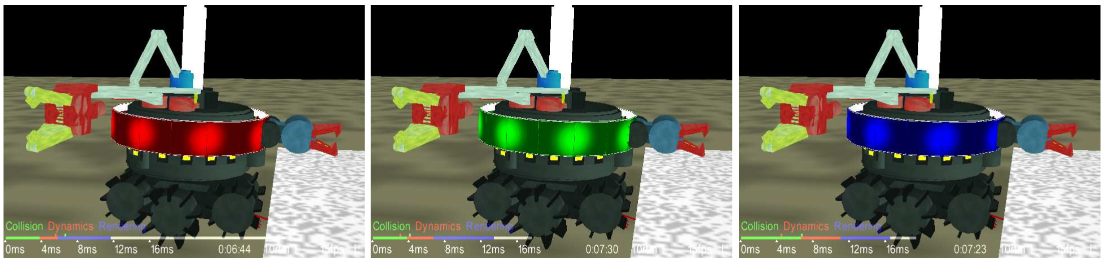


Figure 2.10: Left to right: Detailed s-bot model showing red, green and blue lights, respectively

Table 2.10: Simulation parameters of Light Ring module.

Parameter	**Value**	**Defined in**
number of LED	8×3	modules/LEDModule
LED unit	0.8 mCd	modules/LEDModule
maximum LED intensity	12 mCd (= 15 units)	modules/LEDModule
number of sensors	8	modules/LEDModule
sensor unit	0.1685 mCd/cm^2	modules/LEDModule
maximum sensor unit	700 units	modules/LEDModule
maximum blink frequency	5 Hz	controllers/LEDController

Modeling

The simulated Light Ring is available in the Medium and Detailed s-bot reference models. It is modeled as part of a `<COMPOSITE_GEOMETRY>` in the Turret XML description: this avoids creating an additional dynamic body to define it.

The light intensity of the elements composing a ring are visualized by modifying the graphics color of the objects. Light calculations are based on intensity variables defined in the class `devices/Light`. The class provides methods for calculating the illuminance due to the light source at a some relative position by weighing the source intensity by its inverse squared distance from illuminance position to the source. Light intensities are calculated using *milli-candela* (mCd) as the unit of computation.

Comparison with hardware

Although the hardware light ring module has a maximum blinking frequency of 10 Hz (see standard s-bot API, Appendix A), the simulated Light Ring can only handle a maximum blinking frequency of 5 Hz.

Simulation parameters

The simulation parameters of the Light Ring module are summarized in Table 2.10. The sensor and LED parameters are based from specifications of the hardware.

State variables

Taking into account the description of the Light Ring module, the simulation *state* of the Light Ring can be identified with the following tuple

$$\langle\, r[8], g[8], b[8], blink[8] \,\rangle \tag{2.10}$$

where

$r[i]$	is the red value of LED group i,
$g[i]$	is the green value of LED group i,
$b[i]$	is the blue value of LED group i,
$blink[i]$	is the blink frequency of LED group i

Class description

The Light Ring module is implemented by the class `LightRingModule` (defined in `modules/LEDModule`). Light blinking is controlled by a single thread for all 8 groups and it is implemented by the class `controllers/LightRingBlink`

A UML class diagram for the `LightRingModule` is given in Appendix B.

User programming interface

A user can control and read the simulation state of the Light Ring using the following functions as defined in the standard s-bot API (Appendix A):

```
unsigned getLightSensor(unsigned sensor);
```

This command reads the value of the specified light sensor within the range of $[0;700]$ with 0 representing full darkness and 700 representing full light. It takes one input parameter and it returns one unsigned integer:

`sensor` Light sensor number within the range of $[0;7]$.

```
void getAllLightSensors(unsigned sensors[8]);
```

This command reads the value of all light sensors and dumps them into one array of 8 elements. It takes no input parameters and it returns one array of unsigned integers:

`sensor` Array of light intensities.

```
void setLightColor(unsigned light, unsigned r, unsigned g,
                   unsigned b, BlinkingRate blink);
```

This command sets the color of a light emitter with a specified blinking rate. It takes 5 input parameters and it does not return anything:

`light` Light sensor number within the range of $[0;7]$.

`r` Red light intensity within the range of $[0;15]$.

`g` Green light intensity within the range of $[0;15]$.

`b` Blue light intensity within the range of $[0;15]$.

`blink` Blinking rate expressed in terms of `BLINKING_RATE_xx` values.

```
void setAllLightColor(unsigned r[8], unsigned g[8], unsigned b[8],
                      BlinkingRate blink[8]);
```

This command sets the color of all light emitters. It takes 4 input parameters and it returns nothing:

`r` Array of red light intensities with each value expressed within the range of $[0;15]$.

`g` Array of green light intensities with each value expressed within the range of $[0;15]$.

`b` Array of blue light intensities with each value expressed within the range of $[0;15]$.

`blink` Array of blinking rates with each value expressed in terms of `BLINKING_RATE_XX` values (see `interface/sbot.h`).

```
void setSameLightColor(unsigned r, unsigned g, unsigned b,
                       BlinkingRate blink);
```

This command sets the same color for all light emitters. It takes 4 input parameters and it returns nothing:

`r` Red light intensity expressed within the range of $[0;15]$.

`g` Green light intensity expressed within the range of $[0;15]$.

`b` Blue light intensity expressed within the range of $[0;15]$.

`blink` Blinking rate expressed in terms of `BLINKING_RATE_XX` values (see `interface/sbot.h`).

```
void setLightSamplingRate(SamplingRate rate);
```

This command sets the sampling rate of the light sensors. It takes one input parameter and it returns nothing:

`rate` Sampling rate expressed in terms of `SAMPLING_RATE_XX` values (see `interface/sbot.h`).

In the simulator this command is irrelevant as (for efficiency reasons) the light sensors are updated only when a reading command is issued.

2.2.9 Inclinometer Module

Description

The Inclinometer allows reading of the s-bot pose. Such a device provides information needed for stable navigation on rough terrain.

Modeling

The inclination is defined as the *pitch* (*i.e.*, forward inclination) and *roll* (*i.e.*, lateral inclination) of the tracks system.

Comparison with hardware

No comparison has yet been performed.

Simulation parameters

No model parameters.

State variables

The inclination state of the s-bot can be identified as the following tuple:

$$\langle roll, pitch \rangle \tag{2.11}$$

where

$roll$	is the lateral inclination and
$pitch$	is the forward inclination.

Class description

The Inclinometer is implemented by the class `devices/inclinometer` and it simply returns the pitch/roll values based on the `MdtBody` orientation. The orientation of the `MdtBody` is read using Vortex libraries directly.

User programming interface

Currently, the standard s-bot API (Appendix A) defines two commands regarding the inclination state of the s-bot:

```
void getSlope(int *pitch, int *roll);
```

This command reads the slope pitch and roll angles of a robot. It takes no input parameters and it returns two values:

`pitch` Pointer to the pitch angle expressed within the range of $[-180;180]$.

`roll` Pointer to the roll angle expressed within the range of $[-180;180]$.

```
void setSlopeSamplingRate(SamplingRate rate);
```

This command sets the sampling rate of the slope sensor. It takes one input parameters and it does not return anything:

`rate` Sampling rate expressed in terms of `SAMPLING_RATE_xx` values.

In the simulator this command does not perform anything because the inclination values are not buffered, but read in real-time from the simulation state.

2.2.10 Sound Module

Description

A real s-bot hosts on top of its Turret and at the base of the camera fixture three microphones placed about the central pole at 120° apart from each other. A speaker is available to produce sound. The sound module is intended for simple communication and localization.

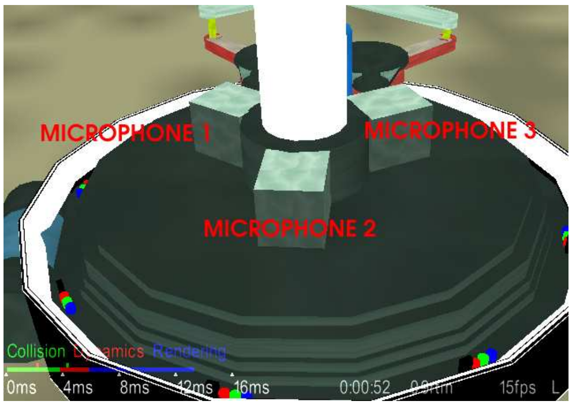


Figure 2.11: At the bottom of the camera pole, three microphones are placed 120° from each other about the central pole.

Modeling

A rigorous sound implementation based on wave PCM is not suitable because the nominal simulation time step 10 ms would only be able to simulate sinusoidal waves with a maximum frequency of 50Hz.

A simplified frequency-based sound module has been implemented in alternative within Swarmbot3d. Each virtual microphone computes its sound intensity by summing up all sound source contributions within a frequency band weighted by the inverse squared distance between microphone and sound source. The implementation is the same as the one discussed in another project publication [1], using the (single source) sound amplitude function

$$A = \sum_{s} \frac{1 - 0.9\,\alpha_S/\pi}{1 + D_S^2/p_0} \tag{2.12}$$

where α_S is the angle between the microphone direction and the direction of the sound source S, and D_S is the geometric distance between microphone and source. The total perceived amplitude (TPA), according to this model is computed using

$$TPA = \sum_{s} \frac{2}{1 + \exp(-\sum_i A_i)} - 1 \tag{2.13}$$

where $\sum_i$ sums over all sound contributions A_i from the other robots i (excluding itself).

Lower and upper frequency limits can be set to determine the band width. Currently, speaker is limited to emit sound of a certain intensity at one single frequency at a time.

Comparison with hardware

At the time of writing the present document, the hardware sound module is still in the early testing phase (Milestone 5, ”Hardware Analysis”). Implementation and API changes of the simulated Sound module are expected as soon as the hardware design is completed.

Simulation parameters

The simulation parameters of the Sound module are summarized in Table 2.11.

Table 2.11: Simulation parameters of sound module.

Parameter	**Value**	**Defined in**
number of microphones	3	modules/SoundModule*
number of speakers	1	modules/SoundModule*
uniform noise	10%	modules/SoundModule*

State variables

Taking into account the description of the simple frequency-based Sound module, the simulation *state* of the Sound module can be identified with the following tuple

$$\langle center, intensity, frequency \rangle \tag{2.14}$$

where

$center$	is the center position of the sound source,
$intensity$	is the current sound intensity,
$frequency$	is the frequency of the sound.

Class description

The Sound module is specified by a base class `modules/SoundModule` and it is implemented by the derived classes `SoundModuleSimple` and `SoundModuleComplex`. The first implementation includes a binary, omni-directional speaker and a number of uni-directional microphones placed in a circle with a radial offset about the s-bot. Currently, only s-bots are checked as sources of sound.

The second implementation in `SoundModuleComplex` allows specification of the placement of microphones and sound sources based on the model descriptions specified in the XML file. Microphone and speaker objects are implemented using the classes `devices/SoundSource` and `devices/SoundSensor`. The functionality is the same as the one offered by the simple implementation.

User programming interface

Currently, the standard s-bot API defines two commands for streaming audio files to the integrated speaker:

```
void playMusic(const char *filename);
```

This command plays the sound specified in filename. It takes one input parameter and it does not return anything:

`filename` String of characters representing the music file name.

```
void stopMusic(void);
```

This command has the effect of simply stopping emitting any music being played. In case no music file is being played, the command has no effect. It does not take input parameters and it returns nothing.

To read the current sound intensity the following function can be used:

```
int getIntensity(int k);
```

This command reads the current sound intensity at microphone $k = 1,2,3$. This function is not yet adopted in the standard API, but is planned in future versions. Details on this interface will be deferred until hardware specifications are known .

2.2.11 Temperature and Humidity Sensors Module

Description

The s-bot hardware has two Temperature Sensors and two Humidity Sensors on each side of the front gripper. The Humidity Sensor provides humidity readings with a time lag of about 4 seconds.

Modeling

Temperature and humidity simulation are outside the capabilities of the current simulator engine, and therefore they have not been modeled in Swarmbot3d. However, these sensors are shortly mentioned in this section for the sake of completeness.

User programming interface

The standard s-bot API provides two commands for reading the temperature and humidity sensors on the left and right sides respectively:

```
void getTemperatureAndHumidityLeft( float *temperature,
                                    float *humidity);
```

This command in case of a real s-bot reads the *left* Temperature and Humidity Sensors. Such a command is currently not implemented in the simulated s-bot, thus its use has no effect within Swarmbot3d. It takes no input parameters and it returns two values:

`temperature` Pointer to the reading of the current sensed temperature expressed in degree Celsius within the range of $[-40;123]$.

`humidity` Pointer to the relative humidity reading within the range of $[0;100]$.

```
void getTemperatureAndHumidityRight( float *temperature,
                                     float *humidity);
```

This command in case of a real s-bot reads the *right* Temperature and Humidity Sensors. Such a command is currently not implemented in the simulated s-bot, thus its use has no effect within Swarmbot3d. It takes no input parameters and it returns two values:

`temperature` Pointer to the reading of the current sensed temperature expressed in degree Celsius within the range of $[-40;123]$.

`humidity` Pointer to the relative humidity reading within the range of $[0;100]$.

Table 2.12: Simulation parameters of the Battery module.

Parameter	**Value**	**Defined in**
nominal voltage	7.2 V	devices/Battery.h
capacity	1400 mAh	devices/Battery.h

2.2.12 Battery Voltage Meter Module

Description

The s-bot hardware is going to be powered by a lithium-ion battery that is placed between the tracks within the Treels body. This battery provides a nominal 7.2V at 1.4 Ah, giving a total capacity of 10Wh (see Deliverable D4, ”Hardware design”). Power consumption of the robot is estimated to be below 5W, ensuring a minimal battery life of two hours.

An accurate battery voltage meter simulation may be useful to develop robot strategies that minimize power consumption. An indication to the current power consumption may be used as a reinforcement feed-back signal for learning algorithms.

Modeling

A simulated Battery Voltage Meter has not yet been fully implemented in the simulator but it has been planned for a future release. To calibrate correctly a Battery Meter, power consumption of each type of actions need to be experimentally measured.

Comparison with hardware

In progress.

Simulation parameters

The simulation parameters of the Battery module are taken from the hardware specifications (Deliverable D4, ”Hardware design”) and they are summarized in Table 2.12.

State variables

Taking into account the description of the Battery module, the simulation *state* of the Battery module can be identified with the following tuple

$$\langle V, c \rangle \tag{2.15}$$

where

V	is the current battery voltage,
c	is the battery capacity.

Class description

A rudimentary battery implementation is provided by the class `devices/Battery` but it has not yet been calibrated against the real hardware specifications.

User programming interface

Currently, the standard s-bot API defines a single command related to the Battery Voltage Meter:

```
unsigned getBatteryVoltage(void);
```

> This command in case of a real s-bot has the effect of providing the current battery voltage expressed in Volts. Such a command is not currently implemented within Swarmbot3d, thus its use has no effect at the moment.
>
> It takes no input parameters and it returns one unsigned integer within the range of $[0;255]$ encoding the current potential available from the batteries. A reading of 0 corresponds to 0 Volts and a reading of 255 corresponds to 10 Volts.

2.3 Swarm Modeling

Description

Up to this point the discussion has considered just one isolated s-bot, however Swarmbot3d has been designed to handle simulations dealing with groups of s-bots. This section is devoted to show how to monitor the status of an entire group of s-bots.

Notice that according to the principles of the swarm intelligence, a swarm should not be modeled as an explicit entity: swarm behaviours should emerge only as a property of the interaction among the units composing a swarm.

Modeling

While a variety of statistical indicators may be used to characterize a group of robots, it is enough at the lowest level to return individual robot positions and headings together with the connectivities of the s-bots within a swarm.

Having this in mind, it is useful to abstract the group of s-bots as a **connected graph** which shows the current swarm configuration. Such a graph is constructed by enquiring all positions, headings and interconnections holding among the s-bots. Figure 2.12 shows such an abstraction of a group of semi-connected s-bots.

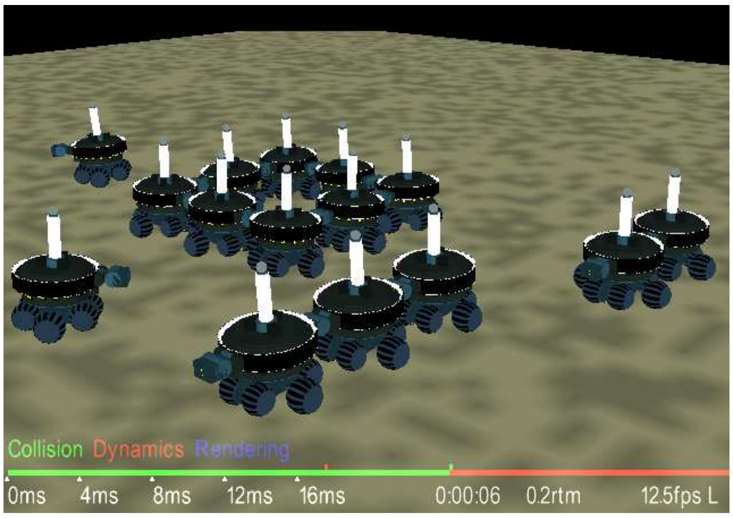
Comparison with hardware

Practically, the global robot positions can be very simply monitored using an overhead camera to track the robots and compute their positions and headings. Often special markers on top of a robot are used to ease image processing, *e.g.*, [4].

All robot positions and headings are directly available within the simulator, thus such an approach is not necessary. However, that could be mimicked by placing a simulated camera above a scene. This could be the method to test tracking algorithms developed for the real s-bot.

Simulation parameters

There are no swarm-bot parameters available because a swarm, according to the basic principles of swarm intelligence, is simply a collection of single s-bots.


Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:06 0.2rtm 12.5fps L

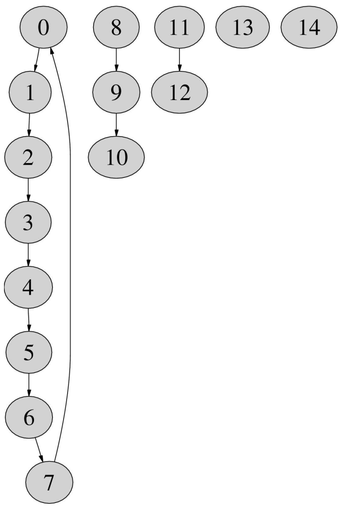


Figure 2.12: Left: Screenshot of multiple s-bots in semi-connected swarm configuration. Right: swarm abstraction using directed graphs.

State variables

Using the abstraction of a connected graph to represent the swarm, the *swarm state* can be identified with the following tuple

$$\langle\, N, pos[N], dir[N], connection[N,N]\,\rangle \tag{2.16}$$

where

N	represents the number of s-bot units,
$pos[i]$	the position of s-bot i,
$dir[i]$	the heading of s-bot i, and
$connection[i,j]$	the connection state between s-bot i and s-bot j.

Class description

A swarm-bot is implemented in the class `robots/SwarmBot` but it is a minimal class that is only intended for monitoring and simulation control of a swarm.

User programming interface

Based on what stated above, the current status of a *swarm* can be retrieved by using the following two commands

```
void SwarmGetPositions (double* pos[3]);
void SwarmGetHeadings  (double* dir[3]);
```

The first command returns a list of the positions of all the s-bots whereas the second one returns a list of all heading directions.

```
int SwarmGetConnectivity (int i, int j);
```

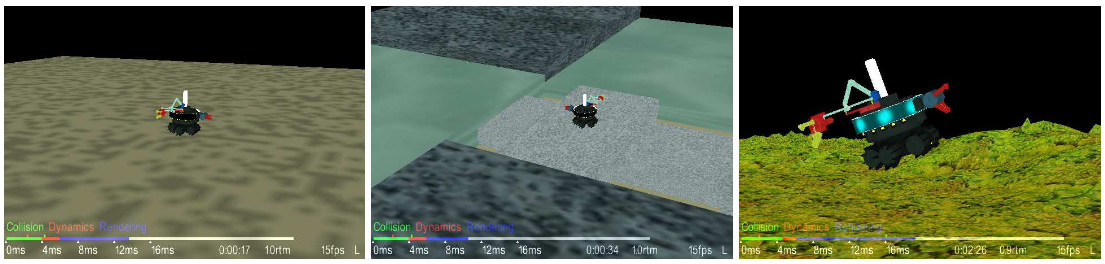

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:17 10rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:34 10rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:02:26 0.9rtm 15fps L

Figure 2.13: Different simulated terrain models. From left to right: (a) s-bot on simple flat plane; (b) s-bot on composite-plane terrain; (c) s-bot on true rough terrain (3D mesh).

This command returns the connection state between two s-bots. It returns 0 if s-bot *i* and *j* are not connected, 1 if they are connected using the front gripper, and 2 if they are connected using the flexible arm.

2.4 Environment Modeling

Besides s-bots, a simulation needs to have environments modeled, too. This last section is dedicated to show how that is achieved within Swarmbot3d.

2.4.1 Terrain Modeling

Description

It was mentioned in an earlier section of this document that Swarmbot3d is designed to deal specially with outdoor rough terrains. It is therefore necessary in a simulation to model simple flat planes as well as rough terrains.

Modeling

To model different environments, it can be employed either a description of a 3D mesh, or that of a simple smooth flat plane. Notice that the latter can also be used to define a piecewise uneven terrain composed of smooth planes at different heights. This is useful for defining terrains with cliffs and sharp falling holes.

A 3D mesh can be realized by loading into the simulator a grey scale bitmap image of 128x128. Notice that the grey scale worked in this case as a sort of depth map whose absolute values were constrained within a given range. Since any grey scale image of the same size can be applied, a user is given the possibility of loading any type of terrain, whose roughness can be increased or decreased by acting on the constraining range mentioned above.

It should be pointed out, though, that the simulation performance is affected by the type of landscape employed. Indeed, it was observed that a swarm simulation using rough terrains gets much slower than the same simulation employing smooth planes. The reason of this behaviour is the increase of collision checking overhead caused by the more complex and uneven geometry of the terrain.

Figure 2.13 shows three terrain models: a simple flat plane, an uneven terrain made of a composition of planes, and a true rough terrain based on a 3D mesh.

Table 2.13: Simulation parameters of a terrain model.

Parameter	**Value**	**Defined in**
terrain type	$\langle plane, composite, rough\rangle$	Resources/Ground/$\langle terrain\rangle$.me
height range[1]	*user defined*	Ground/rough_terrain.me
default size	500×500 cm	Ground/$\langle terrain\rangle$.me
gravity (fast model)	0.0981 m/s^2	World/Fast_World_Settings.me
gravity (others)	9.81 m/s^2	World/World_Settings.me
wheel friction	$12 \cdot 10^4$ dyne	World/World_Settings.me

1) Only for rough terrain type; it determines the ”roughness” of a terrain.

Comparison with hardware

The friction forces between wheels and ground have been experimentally measured by pulling one s-bot using a force meter (”Hardware Analysis”, Milestone 5).

Simulation parameters

Many environment parameters, such as gravity, material properties, and friction, need also to be correctly set in order to have a realistic behaviour. Table 2.13 summarizes the simulation parameters of the terrain model. Parameter values for the simulated gravity have been set to 1/100 of the real gravity for the Fast simulation model, and to real gravity acceleration of 9.81 m/s^2 for all other models. The friction values are based on measured experimental values.

State variables

Synthesizing the discussion outlined above, the parameters describing an environment can be organized into the following tuple:

$$\langle type, roughness, gravity, friction\rangle \tag{2.17}$$

where

type	is the type of landscape (*i.e.*either *smooth* or *rough terrain*),
roughness	describes the percentage of roughness,
gravity	is the gravity pull, and
friction	is the coefficient of friction.

Class description

The terrain is implemented using `McdRGHeightField.h` as defined in the standard Vortex libraries.

User programming interface

The terrain model has no user interface. All parameterization must be specified in the XML file.

Notice that if rough terrain is specified, then the current default grey scale bitmap is going to be used. If a different one is required, then a new name has to be specified into the XML code describing the world scene.

2.4.2 Light and light sensors

Description

Because of the relative ease and simplicity, a variety of robotic experiments use lights in an environment in order to provide directional information to a robot. Thus, it was necessary to include a light model in the simulator.

Modeling

Rigorous light modeling, which also includes shadowing, reflection, and diffusion, is a very complex matter and it has not been implemented in Swarmbot3d. A simplified model is adopted instead. Such a model uses the following assumptions:

- Lights are modeled as isotropic point sources (except for ambient light);
- Sensors are modeled as points;
- Light fall-off is proportional to the inverse squared distance between light and sensor;
- Light shadowing is simplified, *i.e.*, only the direct line of sight is considered;
- No reflection of diffusion is considered;
- Light sensors can define angular sensitivity.

Note that 3D graphics rendering engines, such as OpenGL, do already implement complex lighting calculations. However, only illumination in the visible part of the scene is calculated and therefore this is not suitable for Swarmbot3d because the light needs to be calculated throughout the whole scene for all s-bots.

The **point light** illumination at a certain position is calculated by summing up contributions of all light sources weighted by the inverse squared distance to the source. The sensor response is then calculated by multiplying the illumination value with an angular sensitivity function. The simulator supports a maximum number of 512 point light objects but many light objects are already being used to model the LED of the light ring of one s-bot.

Additionally, **ambient light** is modeled by adding a constant value to the illumination calculations.

Care must be taken not to set the light intensities too high, because it has been observed that light sensors (`LightSensor`) are over-saturated quite easily.

Comparison with hardware

Not yet calibrated with the real robot hardware.

Simulation parameters

Table 2.14 summarizes the simulation parameters of the light model.

State variables

The simulation *state* of a light object can be represented by the following tuple:

$$\langle color, max\_intensity, intensity \rangle \tag{2.18}$$

where

Table 2.14: Simulation parameters of a lighting model.

Parameter	**Value**	**Defined in**
maximum nr. of point lights	512	devices/Light.h
nr. of ambient light	1	devices/Light.h (AmbientLight)
intensity unit	milli-candela (mcd)	devices/Light.h
propagation model	inverse square	devices/Light.cpp
shadowing	yes	devices/LightSensor.cpp
sensor unit	mcd/cm^2	devices/LightSensor.cpp
diffusion/scattering	no	-
reflection	no	-

color	represents the RGB color of the light,
max_intensity	the maximum light intensity,
intensity	the current light intensity,

Class description

The light is implemented by the class `devices/Light` but part of the shadowing calculations are performed by the class `devices/LightSensor`. The latter defines also the angular sensitivity of a sensor.

User programming interface

Obviously, the standard s-bot API does not provide functions to control the environment lights because the robots are not supposed to take control of the light functions.

Ambient and point light have to be controlled directly using the `Light` objects which are accessible through interface functions provided by the class `VortexWorld`:

```
void    setLampIntensity(int nr, float intensity);
double  getLampIntensity(int nr);
void    setAmbientLight(float intensity);
double  getAmbientLight();
```

Chapter 3

Reference Models

This chapter, by using the modules extensively discussed in the previous chapter, presents 4 different s-bot models (Fast, Simple, Medium, Detailed) to be used as reference (Figure 3.1). Each of them has to be viewed as a different approximation of the real robot in the hierarchy of multi-level simulation models. The following sections describe each of them in detail.

3.1 Fast Model

The Fast Model (on left in Figure 3.2) is an extremely simple abstraction of the most salient characteristics describing one s-bot. Its sizes are halved and its mass is 1/20th with respect to its nearest s-bot description (simple model). It is a description meant for simple tests in environment with 1/50th of the real gravity pull. The reduced masses and artificial gravity enables the use of a large simulation time step without getting unstable simulations, at least as along as terrains are defined as planes or at most as composite planes. Such a high time step allows to increase the simulation speed up to 10 times.

Since it implies very little overhead in computation time, its use is mainly intended for quick evaluations requiring large amounts of calculations.

The main characteristics of this model are:

- A robot chassis spherically shaped with two hinged spherical driving wheels and two ball socket jointed caster wheels for maintaining mechanical stability. The chassis and wheels are meant to represent the treels system of a real s-bot.
- A cylindrical turret hinged on top of the chassis equipped with a simple form of IR sensing (sample based look up table) and two spherical wheels hinged to the chassis.

3.2 Simple Model

The Simple Model (second on the left in Figure 3.2) captures like the Fast one just the most important features of a real s-bot while still keeping the simulation speed as fast as possible. Its sizes and its mass approximate sufficiently enough those of a real s-bot.

The Simple Model similarly to the Fast one has the following features:

- A robot chassis spherically shaped with two ball socket jointed caster wheels for maintaining mechanical stability. Even in this case the chassis and wheels are meant to represent just functionally the treels system of the real hardware.

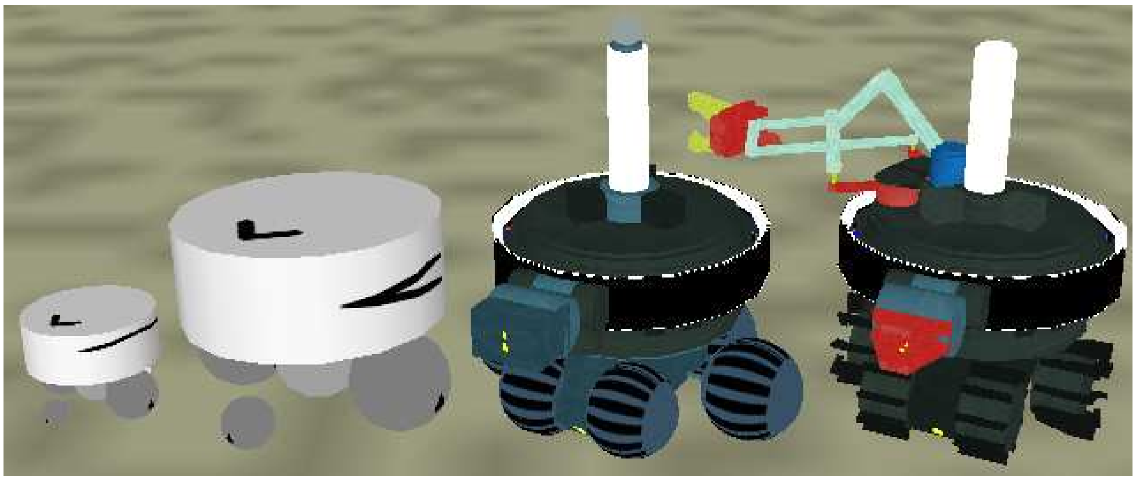


Figure 3.1: All s-bot model approximations: from left to right, Fast, Simple, Medium, Detailed models, respectively.

- A cylindrical turret hinged on top of the chassis and equipped with a simple form of IR sensing (sample based look up table) and two spherical wheels hinged to the chassis.

3.3 Medium Model

The Medium Model (right in Figure 3.2) represents s-bots in a more accurate fashion. The addition of details with respect to the Simple model is particularly important in the case of navigation on rough terrain.

Similarly to the Simple Model, the Medium one is made up of two sub-systems (Treels and Turret), however each of them is defined in a more accurate and detailed fashion. The main characteristics of this model description are:

- A detailed s-bot chassis (Treels subsystem).

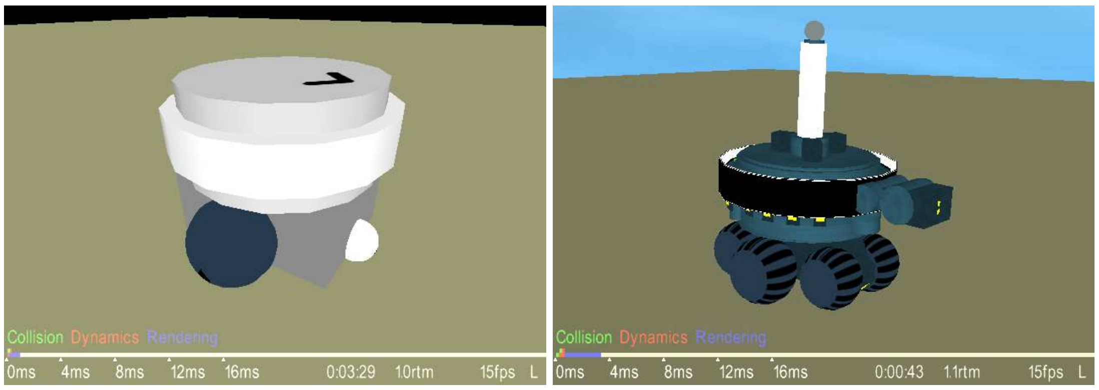

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:03:29 1.0rtm 15fps L
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:43 1.1rtm 15fps L

Figure 3.2: Fast/Simple S-Bot Model (left) and Medium S-Bot Model (right).

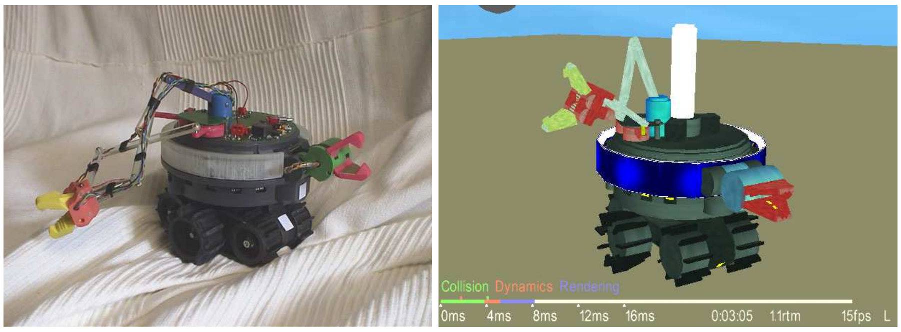

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:03:05 1.1rtm 15fps L

Figure 3.3: Real S-Bot (left) and Detailed Model (right).

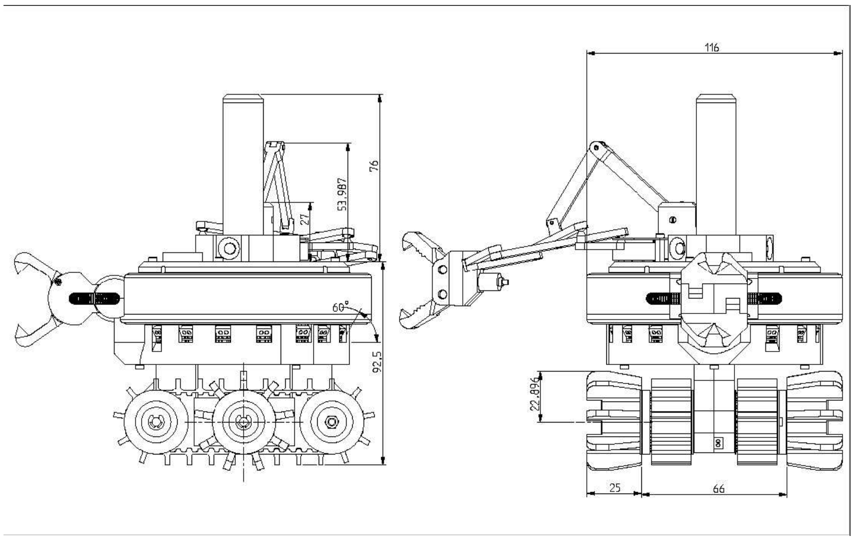


Figure 3.4: Real s-bot blueprint.

- Six spherically shaped wheels hinged to the chassis as in the real robot.
- A detailed description of the Turret subsystems comprehensive of IR, sound, and camera sensors.
- A magnet-like gripper shaped as a box capable of dynamically realizing connection and disconnection.

3.4 Detailed Model

The Detailed Model (Figure 3.3) provides an accurate representation of a real s-bot. This model description is used in situations where a close interaction between two robots is required as in the case of gripping.

This model replicates in details the geometrical blue prints of the real hardware (Figure 3.4) as well as masses, centre of masses, torques, acceleration, and speeds (cf. Deliverable 4, ”Hardware Design”).

The Detailed Model is made up of four mechanical modules: Treels, Turret, Front Gripper Arm, and Flexible Gripper Arm. Its main characteristics are as follows:

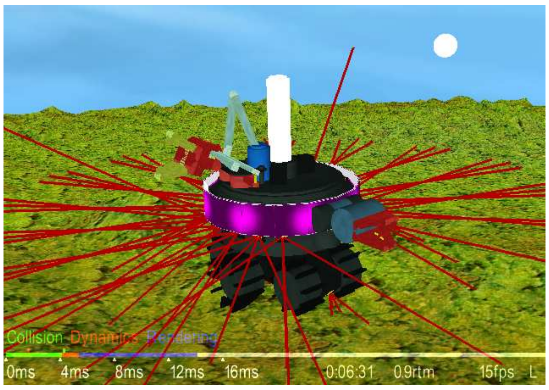


Figure 3.5: Detailed S-Bot Model navigating on rough terrain using the lateral IR sensors and emitting blue light.

- A detailed chassis description comprehensive of 4 IR sensors (2 at the bottom, 1 in front and 1 on the back).
- Six teethed wheels, three on each side, with the two middle ones located slightly outward as in the hardware s-bot.
- Detailed Turret representation as in the Medium Model.
- One Gripper Arm hinged on the front of the turret and endowed with two teethed jaws.
- One scissors-like Flexible Gripper Arm attached through three hinges to the s-bot's body and endowed with two teethed jaws.

The Detailed Model closely matches the s-bot hardware, however it lacks the caterpillar-like rubber teethed band joining the internal wheels of each track. The simulation of this track proved to be too computationally expensive providing only a negligible gain in the realism of simulating the real hardware.

The rotating Turret sub-system consists of a cylindrical disc encasing the gears driving its own rotation about its central axis and the gears driving Front and Flexible Gripper Arms. Within the Turret, there are also 15 lateral IR sensors which have been modeled as an array of probing rays, each with a given direction (Figure 3.5).

The surrounding ring geometry acts as a passive attachment device for the fixed grippers on the Front Arm. The three RGB colours of each of the 8 light sensors can be controlled independently so as to act as a light emitting signalling device for other s-bots, *e.g.*, see Figure 3.5.

Each s-bot is provided with an omni-directional camera system placed within the tube protruding from the centre of the Turret. The base is equipped with three sound detectors (Figure 3.3). A virtual omni-directional camera approximates the spherical mirroring viewed by the real camera placed at the bottom of the protruding tube. The sound detectors at the base of the tube are modelled as point-like microphones with a cardioid sensitivity pattern.

The Front Gripper sub-system is a accurate model of a jaw gripper with protruding teeth. The geometry of these gripper teeth matches exactly the geometry of the light ring of the s-bot Turret so that a physical connection can be achieved by opening and closing these jaws (Figure 3.6). The lower jaw has two simulated optical barrier sensors which can be used for detecting objects in its close proximity.

The Flexible Side Arm sub-system is modeled as a scissor-like geometry extendible in 3D by using its three inner joints (Figure 3.7). At the utmost tip of such a structure is located the second s-bot's gripping device. Connection to other s-bots occurs simply by closing such a gripper.

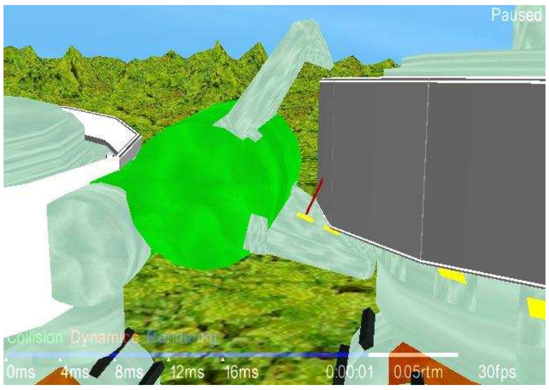


Figure 3.6: Detailed s-bot model in the act of gripping a fellow s-bot.

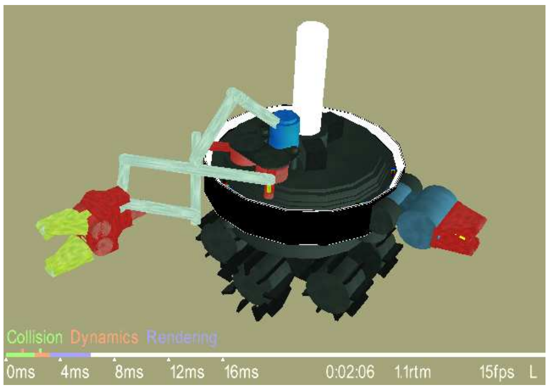


Figure 3.7: Flexible arm extending out from a detailed s-bot model.

Chapter 4

Validation Experiments

This chapter presents the experiments carried out for validating the simulation environment Swarmbot3d against the real hardware (**Section 4.1**) and for evaluating the various levels of abstractions of one s-bot with respect to the aforementioned validation (**Section 4.2**). The last part of the chapter shows how these hierarchical abstractions can become useful for simulating the navigation of one s-bot on a composite terrain made of smooth planes, gaps, slopes, and rough surfaces using a dynamic model changing technique (**Section 4.3**).

Notice that part of these experiments have already been presented in an earlier document of this project (Deliverable D8, "Swarm-bots Prototype and Evaluation"), however they are reported here too for the sake of completeness.

4.1 Validation Against Hardware

This section describes three experiments carried out to calibrate the behaviour of the simulated s-bot with that of the real one [7]. The first concerns climbing a slope (Section 4.1.1), the second passing a gap (Section 4.1.2), whereas the last one concerns climbing a step with two connected robots (Section 4.1.3)

4.1.1 Real vs. Simulated Slope Climbing

To achieve a climbing calibration both real and simulated s-bots were placed on a similar inclined surface with the same friction and both of them were tested against the ability first to stay still and then to move uphill.

The purpose of this test was to match the ability of one simulated s-bot with that of the real one either by successfully climbing hills and slopes or by failing the task altogether exactly as it would happen in reality. This is crucial in view of navigation tasks to be performed on rough terrain. The test was then repeated with two s-bots connected into a chain by means of the front gripper of the second s-bot.

By experimenting with different slopes, it emerged that the maximum inclination angle manageable by a group of connected s-bots is considerably larger than that reachable by a single s-bot (Table 4.1).

To simulate the real experiments reported above, a rigid polystyrene plastic has been used. The ground friction within the simulator was set to the same value measured in reality (1.2 N).

4.1.2 Real vs. Simulated Gap Passing

This experiment was set up in order to calibrate how s-bots manage to deal with gaps of various widths. By running experiments with the real robots, it was established that a single s-bot was able to carry out a

Table 4.1: Comparison between real and simulated s-bot behaviour both for single and connected s-bots placed on a slope of various inclination angles: ’full’ stands for full maneuverability, ’limit full’ stands for limit of full maneuverability, ’topple b/w’ stands for topple backwards, and ’limit stable’ stands for limit of stable position.

	Slope	**Hardware Robot**	**Simulated Robot**
Single S-Bot	10°	full	full
	20°	diffi cult	full
	30°	topple b/w	topple b/w
two s-bots connected	29°	limit full	limit full
	58–72°	limit stable	limit stable

Table 4.2: Comparison between real and simulated s-bot behaviour with respect to passing a gap: pass stands for successful traversing, whereas falls inside for failing to overcome it.

Gap (mm)	**Hardware Robot**	**Simulated Robot**
30	pass	pass
45	pass	pass
55	fall inside	fall inside

successful traversal up to a maximum gap of about 45 mm. Beyond this width, the robot was every time falling in it and there getting stuck.

After calibrating the detailed simulated s-bot, the same experiment was run within Swarmbot3d. By widening at each trial the size of the gap, it was observed that the behaviour of the simulated robot was properly replicating that of the real one. Results are synthesized in Table 4.2.

Figure 4.1 shows a snapshot comparison between the real and simulated s-bot for a gap of 55 mm: in both cases, the robot fails to pass and falls in the gap.

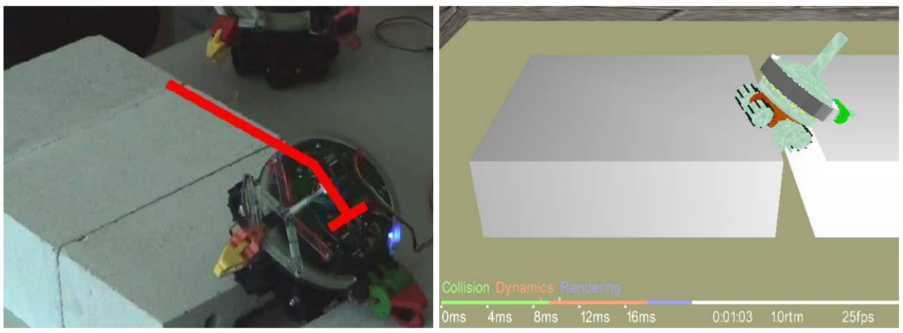


Figure 4.1: The real robot in reality compared with its detailed approximation within the simulator: both fall in a gap 55 mm wide.

Table 4.3: Maximum step climbing width for one single s-bot going backward and forward.

step height (mm)	**simulated s-bot**		**real s-bot**	
	backward	forward	backward	forward
≤ 15	pass	pass	pass	pass
16	fail	”	fail	”
23	”	”	”	”
24	”	fail	”	fail

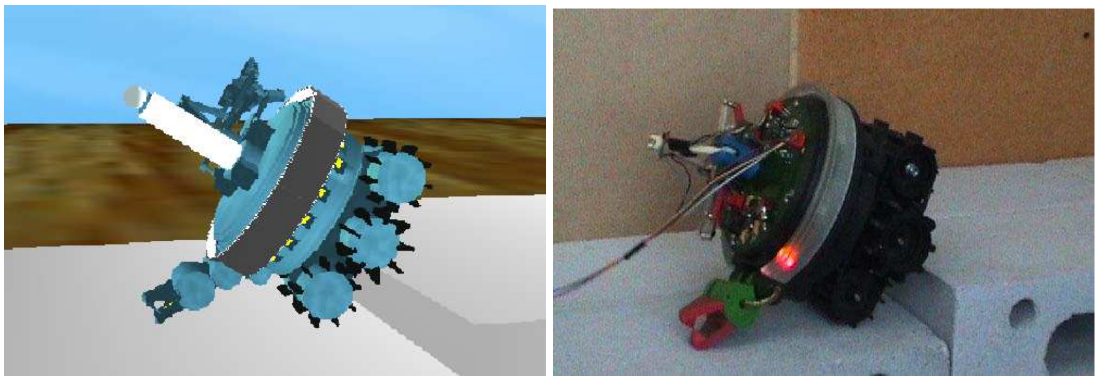


Figure 4.2: Equivalent behaviour of simulated and real s-bot with respect to a step size of 32 mm.

4.1.3 Real vs. Simulated Step Climbing

This experiment was set up in order to calibrate the behaviour of one simulated s-bot with that of the real one with respect to the task of climbing a step.

The step climbing experiment was split in two different parts: the first one calibrates a single s-bot, whereas the second one calibrates two s-bots connected in a chain.

Single S-Bot

One robot was placed close to a step and then moved towards it. The operation was carried out with both forward and backward motion. Each time the test was performed, the step height was increased of 5 mm.

At the end of this sequence of tests, it emerged that the maximum step size which a single s-bot was able to cope with was 15 mm going backward and 23 mm going forward (Table 4.3).

The difference between the two behaviours both in the simulated robot and in the real one is due to the center of mass of the turret which is 4 cm off centered towards the gripper. The reason for this stems from the mass of the gripper motors which are placed right behind it. It is in any case interesting to notice that both simulated and real s-bots produce equivalent response.

Figure 4.2 shows that at a step of 32 mm, which is the height at which two connected robot can pass (see next section), one single s-bot going backward topples over both in the simulated world and in reality.

Two Connected S-Bots with Gripper

The second part of the step passing experiment was carried out with two s-bots connected in a chain. To do this, two disconnected s-bots were first placed one after the other along an orthogonal direction with respect

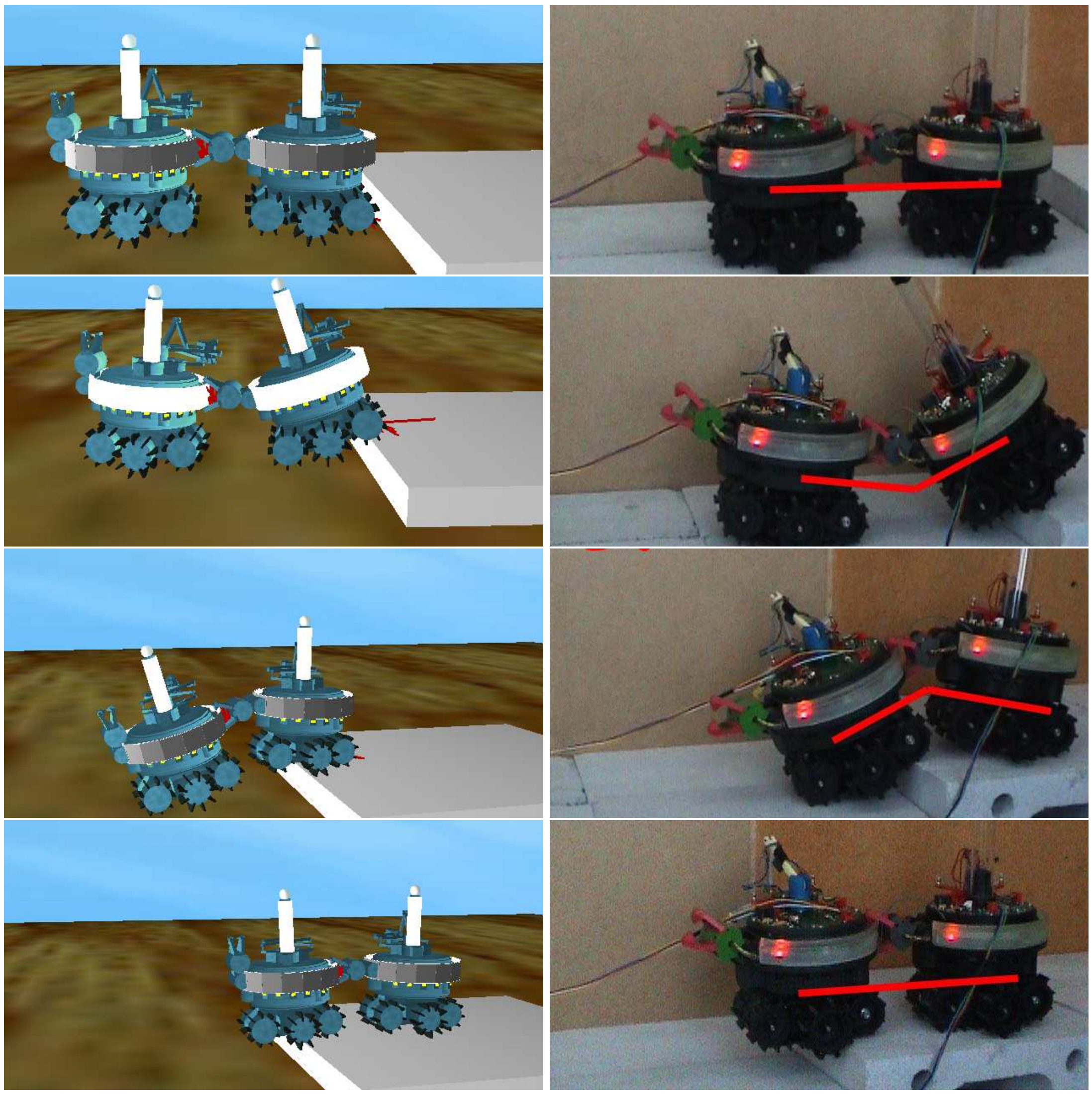


Figure 4.3: From left to right: the different phases of the step passing for the limit step of 32 mm. The pictures on the first column refer to the simulated s-bot and those on the second column to the actual s-bots.

Table 4.4: Comparison of features among Fast, Simple, Medium, and Detailed s-bot simulation models. Yes and no respectively indicate the presence or absence of a particular feature in the model.

Model	**Fast**[1]	**Simple**	**Medium**	**Detailed**
driving wheels #	2	2	6	6
IR proximity sensors	yes[2]	yes[2]	yes	yes
ground sensors	no	no	yes	yes
camera	no	no	yes	yes
sound module	yes[2]	yes[2]	yes	yes
front gripper	no[3]	no[3]	yes[4]	yes
flexible side arm	no	no	no	yes
dynamical bodies #	6	6	8	23
typical RTM	950[1]	94	31	3
time step	100[1]	10	10	10

1) Measurement taken with 1/50th of gravity and 1/20th of mass.
2) Using sample-based look-up table.
3) No physical gripper model: connections are possible using virtual links.
4) Coarse version, i.e. sticking box.

to the step, and then let them connect and move backward towards the step. By varying the height of the step at each climbing, it was observed that the two robots were able to overcome up to 32 mm, exactly in accordance with what experienced with the real s-bot.

Figure 4.3 shows four stages in passing the limit step of 32 mm. First, the two s-bots approach the step in backward formation. Second, as soon as the first robot senses the step with its back-facing ground sensor, it starts lifting itself using its connected front gripper arm. During traversal, the robot bends its gripper arm in opposite direction (downward) pushing itself up. Finally, the first robot continues its backward motion and pulls in this way the second one over the step to complete the procedure.

4.2 S-Bot Abstraction Evaluation

This section evaluates the different abstraction levels available for representing one simulated s-bot. These different approximations (Fast, Simple, Medium, and Detailed) are ordered in an increasing level of refinement.

4.2.1 Efficiency Tests

The first test reported here establishes the computational overheads introduced by the use of a specific abstraction. Table 4.4 summarizes the differences among the models tested with a single s-bot on a flat plane. These results are indicated in Real Time Multiplier (RTM) values which represent how fast real time can be simulated (*e.g.,* RTM = 100 means that a simulation is 100 times faster than real time). RTM values depend, of course, on the available computing hardware; specifically the tests reported below were carried out on a dual 3.06GHz Xeon PC with one nVidia QuadroFx 1000 graphics card.

4.2.2 Hierarchical Models' Comparison Tests

The second series of tests reported in this section shows how the various s-bot abstractions compare with the Detailed model which is used as a reference, given its established calibration with the real robot.

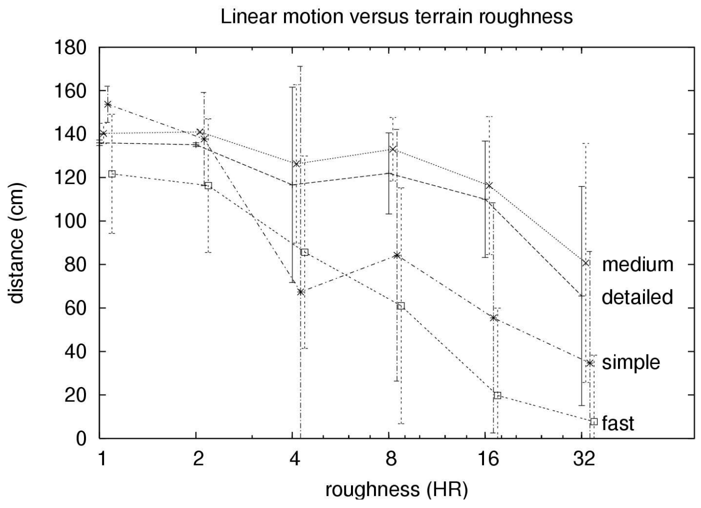


Figure 4.4: Comparison of linear motion errors of the three s-bot simulation models with respect to different terrain roughness.

Motion Comparison

This experiment compares how different models differ during forward linear motion on terrains with different levels of roughness. To do so, each simulation model was placed randomly on a terrain and then assigned a random heading. The terrain roughness was controlled within the simulator by specifying a height range (HR) parameter. The HR values correspond to descriptive values: 1 for almost flat, 2 for minimally rough, 4 for little rough, 8 for mildly rough, 16 for rough, and 32 for very rough.

Given the reduced size of the Fast model, the terrain used for that s-bot abstraction was scaled down by half from its original size in order to retain comparable conditions.

Figure 4.4 shows the motion errors which are obtained by letting each s-bot model run at medium speed (15.24 cm/s) for 10 seconds. The vertical axis plots the projection of the travelled distance onto the randomly selected initial direction. Depending on the terrain roughness this distance decreases because the s-bot is not able to retain a straight course. The small differences in distance on flat terrain are caused by calibration errors of the velocity due to differences in the wheel diameter among the various s-bot approximations.

Figure 4.4 shows that the rough terrain motion of the Medium model closely follows the behaviour of the Detailed s-bot model: the constant offset is due to differences in wheel size (see above). Both Simple and Fast models quickly fail to retain linear motion even on minimally rough terrains. Since the Detailed model replicates quite closely the behaviour observed on the real robot, this suggest that also the Medium one can reasonably approximate it, at least as far as pure locomotion on rough terrain is concerned. The Fast and Simple Models should not be used for experiments using rough terrain.

4.2.3 Gap Passing

This particular experiment examines the problem of gap passing (Figure 4.5). The various s-bot approximations are compared with the detailed one, which, given its close similarity with the real robot, is used also in this case as a reference.

The gaps are defined by means of two planar surfaces placed at a given distance. Table 4.5 summarizes

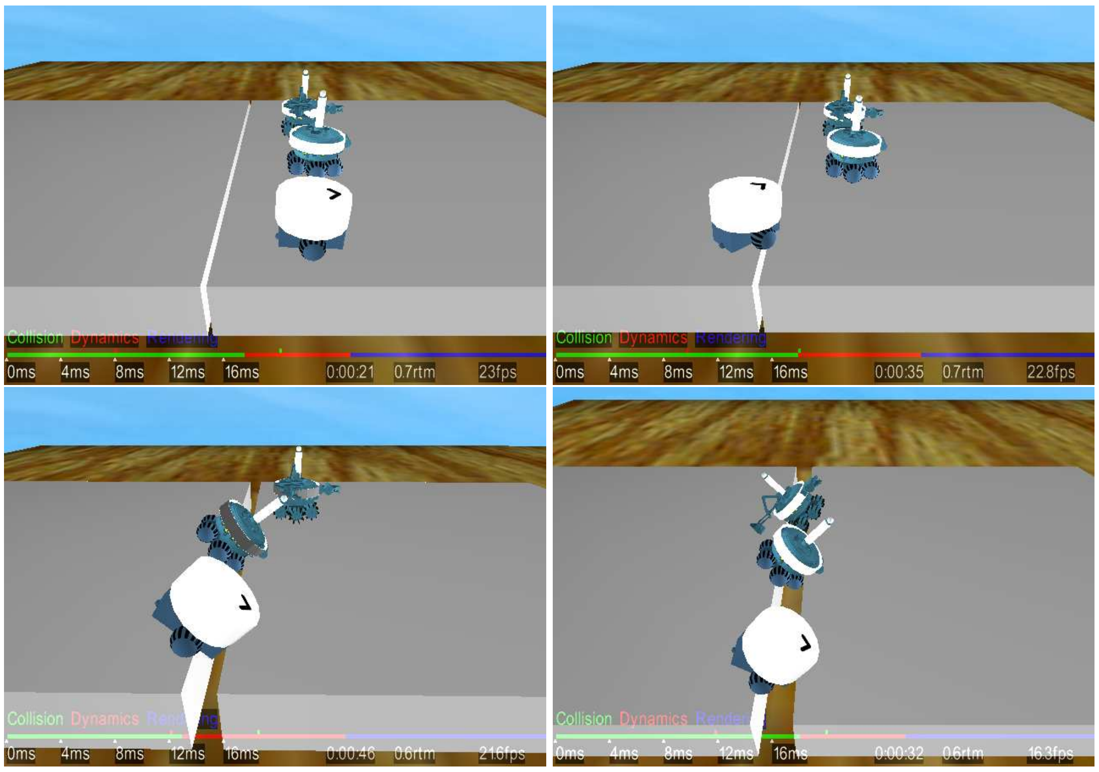

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:21 0.7rtm 23fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:35 0.7rtm 22.8fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:46 0.6rtm 21.6fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:32 0.6rtm 16.3fps

Figure 4.5: From left to right, and from top to bottom: the 3 s-bot models while traversing a gap of 8, 9, 40 and 60mm, respectively.

Table 4.5: Gap passing comparison of the four s-bot approximations used.

Gap (cm)	**Fast**	**Simple**	**Medium**	**Detailed**
3.0	pass	pass	pass	pass
4.0	“	“	“	“
4.5	“	pass (70%) stuck (30%)	pass (90%) stuck (10%)	“
5.0	“	stuck	stuck	“
5.7	“	“	“	pass (50%) stuck (50%)
6.0	“	“	“	“
6.5	stuck	“	“	“

the tests obtained by varying the size of such a gap. Each table entry corresponds to the modal value of 12 observations. All experimental tests were carried out by using a low speed of 2.8 cm/s in order to exclude effects from finite inertia, i.e. that the robot would pass the gap merely due to a high speed.

4.2.4 Slope Tackling

This experiment reports how the various s-bot models compare when they encounter a slope (Figure 4.6). The navigation skills of the detailed model are once again used as a reference.

To quantify how the 4 models behave with respect to a sudden change in the evenness of the ground, three simulated s-bots, each expressed at a different level of refinement (Detailed, Medium, and Simple), were placed on a plane and then let them move forward until an uphill slope was encountered. The fast model had to be tested separately and alone, since it was meant to run in an environment with 1/50th of gravity pull and with a time step 10 times higher (100ms).

The ability of each s-bot model to cope with a slope was observed as soon as each of them encountered a variation in the ground.

The maximum tilting angle each model was able to cope with was measured by progressively changing the inclination angle of the slope with respect to the ground plane. The speed of all s-bots was also in this case set to 2.8 cm/s for the same reason pointed out in the gap passing experiment. Results are summarized in Table 4.6: each entry corresponds to the modal value of 12 observations.

Downhill slopes are much less of a problem, since gravity, by forcing contact between s-bots and ground, provides a mechanical aid to the motion. All models, in fact, successfully managed this type of hindrance test up to about 0.5 radians. When slopes become steeper, all s-bots topple forward exactly as the real robot does.

4.3 Simulation of Navigation on Composite Terrain using Dynamic Model Switching

The aim of this experiment is to show a practical use of the hierarchy of models representing one s-bot. The problem considered here is the navigation of a single s-bot onto a complex terrain presenting different levels of difficulty in different stages. The hurdles are represented by three gaps, one slope, and two rugged areas.

For this experiment, a variant of the Simple Model, called Basic Model, is introduced. This model differ from the Simple one in the following points: (a) presence of a magnetic-like sticking device similar to that available for the Medium model, (b) ray traced IR sensors, (c) offset center of mass matching that

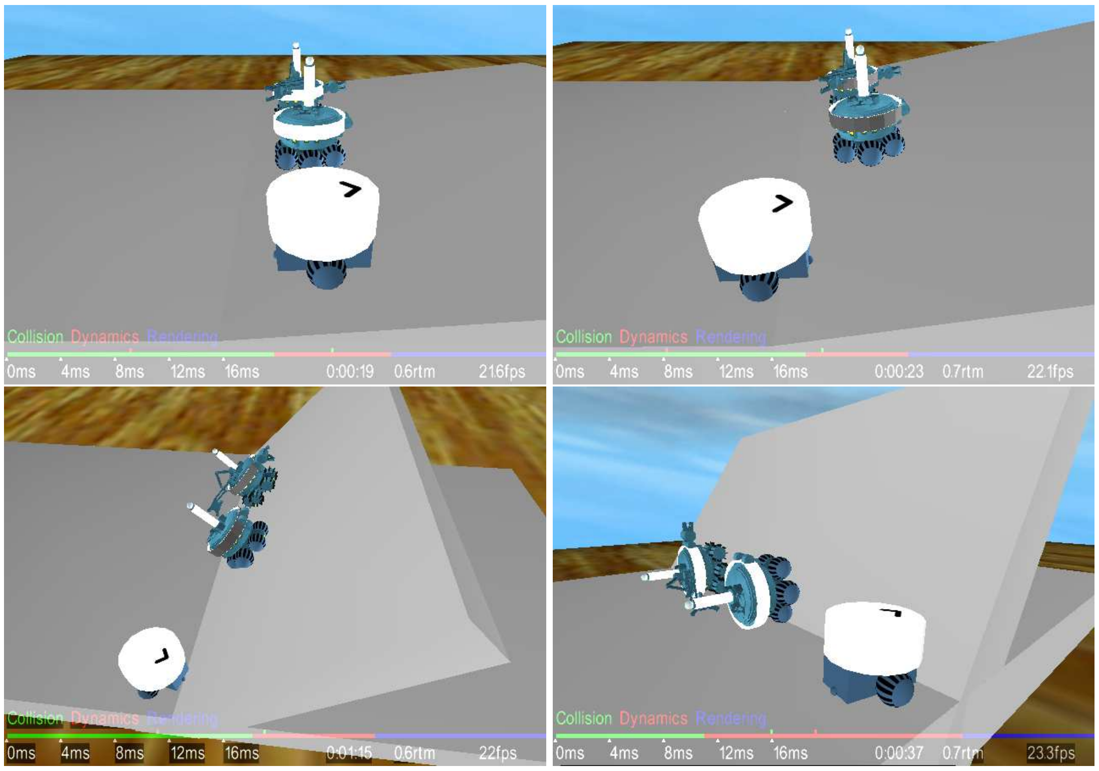

Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:19 0.6rtm 21.6fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:23 0.7rtm 22.1fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:01:15 0.6rtm 22fps
Collision Dynamics Rendering
0ms 4ms 8ms 12ms 16ms 0:00:37 0.7rtm 23.3fps

Figure 4.6: From left to right: the 3 s-bots models tested for upward slope with inclination angle 0.1, 0.2, 0.8 and 1.0 radians, respectively.

Table 4.6: Slope climbing comparison of the three s-bot approximations used: 'climb' stands for successful climb, 'c/c' stands for change course at the foot of the slope, and 'topple' means the s-bot toppled over backwards.

Angle (rad)	**Fast**[1]	**Simple**	**medium**	**detailed**
0.02	climb[2]	climb[3]	climb	climb
0.04	"	stuck (60%) c/c (40%)	"	"
0.20	c/c	stuck	climb	climb
0.70	"	"	"	"
0.80	topple	"	slipping	slipping
0.90	topple	"	slide back	topple
1.00	stuck	"	topple	topple

1) Measurement taken in low gravity environment and with a high time step.
2) Slope is overcome but course is changed slightly.
3) Slope is taken but the initial direction is lost: s-bot changes course randomly.

of the higher level approximations, and finally (d) a more accurately calibrated treels system. The intention is to use the Basic model instead of the Simple one in future experiments involving connected swarm configurations. The characteristics of this model allow it to replace quite well the Medium and Detailed Models on the fly, that is during a simulation [8].

The simulation of this task is first carried out using the Detailed Model and then repeated using a technique of *dynamic model exchange*. Such a simulation technique consists in letting one s-bot navigate along a path and in switching its model representation on-the-fly according to the kind of terrain encountered (Figure 4.7).

The simulation starts with the Basic model on a flat plane. As long as the s-bot moves on flat ground, this simple robot representation is retained. As soon as a harder terrain is encountered (*e.g.*, smaller gaps up to 5 cm, a slope, or medium rough terrain), the representation is switched to the Medium model. When also this one is not sufficient anymore (*e.g.*, gaps larger than 5 cm, or very rough terrain situations), the Detailed model description is then used. As soon as a smooth plane is detected again, the simulator switches s-bot representation back to the Basic model, and so on. The gain obtained using the model changing technique is expressed as a reduction of total simulation time.

The diagram at the top in Figure 4.8 plots the aforementioned complex path followed by the detailed model repeated 10 times. The diagram at the bottom in the same figure plots the same path carried out using the model switching technique. The two plots show that using model switching the simulated robot reaches the final position of the detailed robot within an acceptable range. It can be observed that the detailed model is not very accurate in making right angle turns due to occasional slipping of the teethed wheels. The other two models do not have this problem, thus the plots using model switching show a more compact clustering.

The simulated real time for completing complex navigation task was 59 real world seconds. The total simulation time required using the detailed model alone was about 64 seconds, whereas using the model switching technique was less than 28 seconds. This means that using model switching allows to reduce simulation time by more than half.

It should be pointed out, however, that any gain using the technique presented here is obviously dependent on the combination of terrain types. In the example discussed above, the more the environment allows to use the simpler models, the more gain is expected in terms of simulation time.

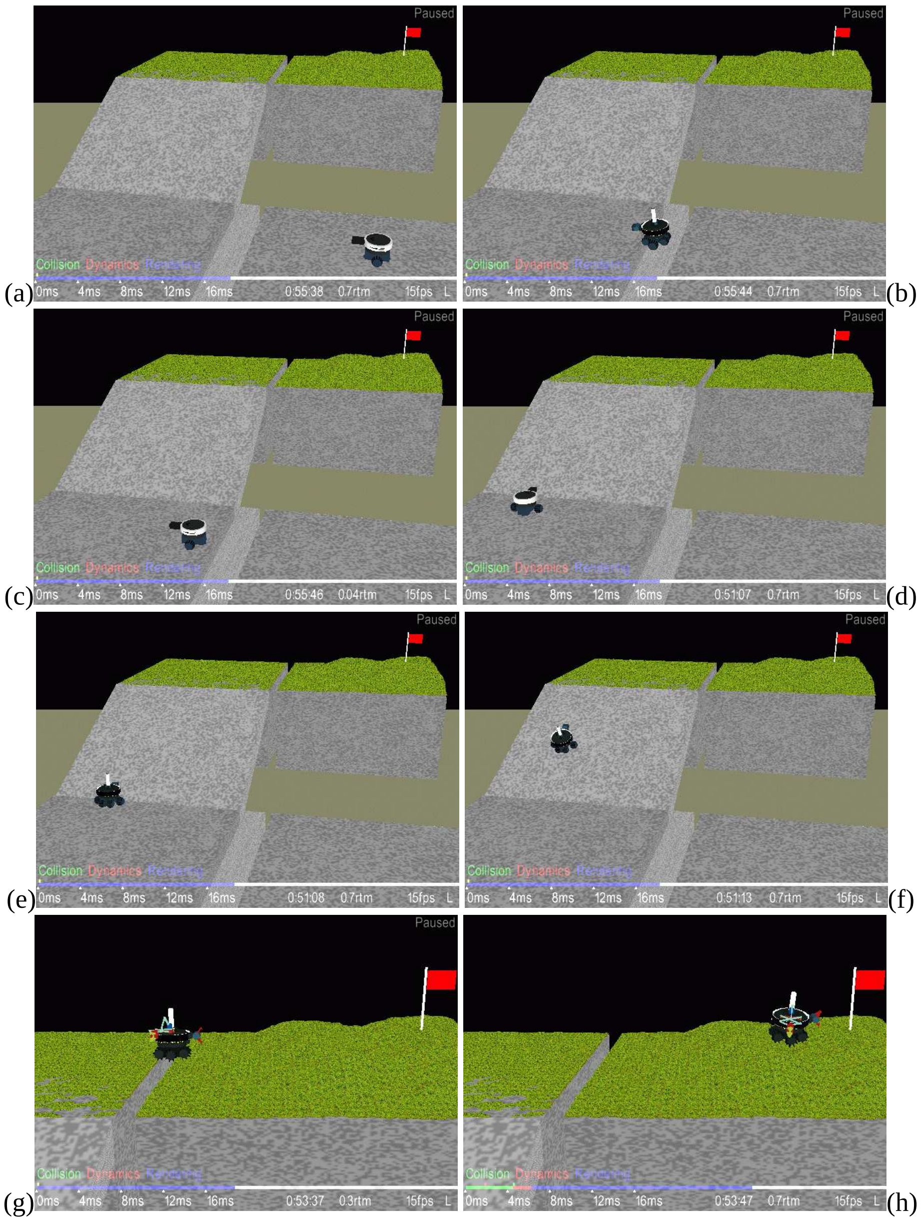
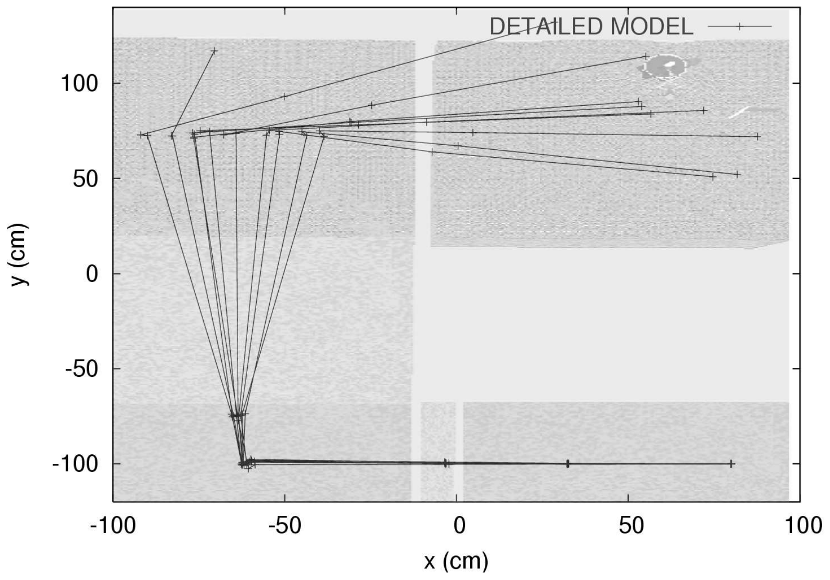


Figure 4.7: Simulation of one s-bot navigating onto different terrains while switching its model representation. (a) The simulation starts with the Simple model; (b) at the trench the model switches to the Medium; (c) right after the trench it switches back to Simple; (d) just before the slope the switch from Simple, (e) to Medium; (f) on the slope the Medium representation; (g) switch to Detailed at the large trench; (h) remain Detailed on the rough terrain.



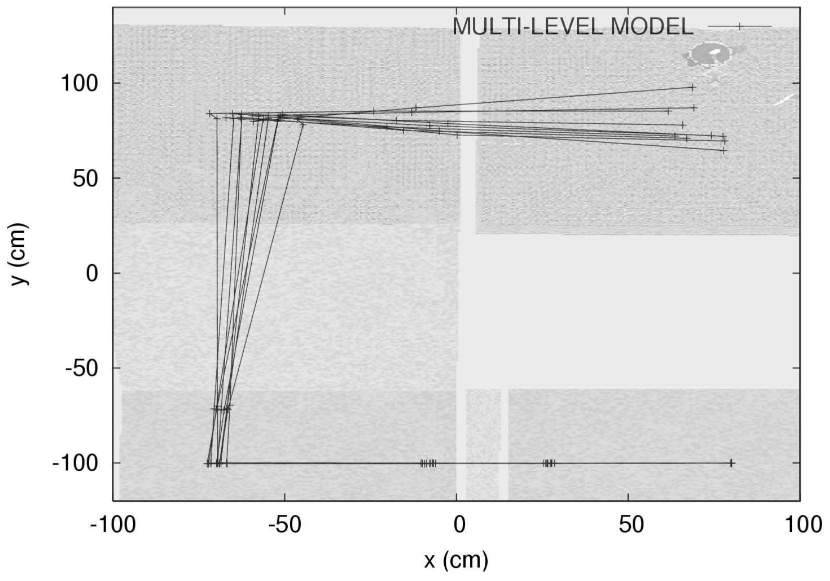


Figure 4.8: Tracked path of the simulation using just the detailed s-bot model (top) and that of the simulation using the model switching technique (bottom).

Acknowledgments

The work presented here is part of the SWARM-BOTS project, which is funded by the Future and Emerging Technologies programme (IST-FET) of the European Community under grant IST-2000-31010 and by the Swiss Government under grant 01.0012.

The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data published here.

Bibliography

[1] G. Baldassarre, S. Nolfi, and D. Parisi. Evolving mobile robots able to display collective behaviour. *Artificial Life*, 9:255–267, 2003.

[2] Luiz Chaimowicz, Mario Campos, and Vijay Kumar. Simulating Loosely and Tightly Coupled Multi-Robot Cooperation. In *Anais do V Simpósio Brasileiro de Automação Inteligente (SBAI)*, Canela, RS, Brazil, September 2001.

[3] Bob Diamond. Concepts of Modeling and Simulation. http://www.imaginethatinc.com/pages/sim_concepts_paper.html, 2002.

[4] H. Lund, E. Cuenca, and J. Hallam. A Simple Real-Time Mobile Robot Tracking System. Technical Paper 41, Department of Artificial Intelligence, University of Edinburgh, 1996.

[5] O. Michel. Webots: Symbiosis between Virtual and Real Mobile Robots. In *Proceedings of the First International Conference on Virtual Worlds, VW'98*, pages 254–263, Paris, France, 1998. Springer Verlag.

[6] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm Simulation System, a Toolkit for Building Multi-Agent Simulations, 1996.

[7] Francesco Mondada, Giovanni C. Pettinaro, André Guignard, Ivo W. Kwee, Dario Floreano, Jean-Louis Deneubourg, Stefano Nolfi, Luca Maria Gambardella, and Marco Dorigo. SWARM-BOT: a New Distributed Concept. *Autonomous Robots*, 2003. To appear soon.

[8] Giovanni C. Pettinaro, Ivo W. Kwee, and Luca M. Gambardella. Acceleration of 3D Dynamics Simulation of S-Bot Mobile Robots using Multi-Level Model Switching. In *Submitted to the IEEE International Conference on Robotics and Automation (ICRA '04)*, New Orleans, Louisiana, U.S.A., 26th April–1st May 2004.

[9] Giovanni C. Pettinaro, Ivo W. Kwee, Luca M. Gambardella, Francesco Mondada, Dario Floreano, Stefano Nolfi, Jean-Louis Deneubourg, and Marco Dorigo. Swarm Robotics: a Different Approach to Service Robotics. In *Proceedings of the 33rd International Symposium on Robotics (ISR '02)*, pages 71–76, Stockholm, Sweden, 7th–11th October 2002.

Appendix A

Standard S-Bot API

```
/*
sbot.h : c interface to the s-bot robot
(c) 2003 Stephane Magnenat, LSA 2, EPFL, SwarmBot Project
*/

#ifndef __SBOT_H
#define __SBOT_H

/* Set up for C function definitions, even when using C++ */
#ifdef __cplusplus
extern "C" {
#endif

/** Sample sensor 50 times per second (50 Hz) */
#define SAMPLING_RATE_50 1
/** Sample sensor 33 times per second (33 Hz) */
#define SAMPLING_RATE_33 2
/** Sample sensor 20 times per second (20 Hz) */
#define SAMPLING_RATE_20 3
/** Sample sensor 10 times per second (10 Hz) */
#define SAMPLING_RATE_10 4
/** Sample sensor 5 times per second (5 Hz) */
#define SAMPLING_RATE_5 5
/** Sample sensor 2 times per second (2 Hz) */
#define SAMPLING_RATE_2 6
/** Sample sensor one time per second (1 Hz) */
#define SAMPLING_RATE_1 7
/** Sample sensor every two seconds (0.5 Hz) */
#define SAMPLING_RATE_0_5 8

/** The sampling rate. Can take values SAMPLING_RATE_xx */
typedef unsigned SamplingRate;

/** Do not blink leds*/
#define BLINKING_RATE_NO_BLINK 0
/** Blink leds 10 times per second (10 Hz) */
#define BLINKING_RATE_10 1
/** Blink leds 5 times per second (5 Hz) */
#define BLINKING_RATE_5 2
/** Blink leds 3.3 times per second (3.3 Hz) */
#define BLINKING_RATE_3_3 3
/** Blink leds 2.5 times per second (2.5 Hz) */
#define BLINKING_RATE_2_5 4
/** Blink leds 2 times per second (2 Hz) */
#define BLINKING_RATE_2 5
/** Blink leds one time per second (1 Hz) */
#define BLINKING_RATE_1 6
/** Blink leds every two seconds (0.5 Hz) */
```

```
#define BLINKING_RATE_0_5 7
/** Blink leds every three seconds (0.33 Hz) */
#define BLINKING_RATE_0_33 8
/** Blink leds every four seconds (0.25 Hz) */
#define BLINKING_RATE_0_25 9
/** Blink leds every five seconds (0.2 Hz) */
#define BLINKING_RATE_0_2 10

/** The blink rate. Can take values BLINKING_RATE_xx */
typedef unsigned BlinkingRate;

/** Initialize the s-bot. Request I2C low level interface and set
default values. This function takes two arguments : they are
the callback function when an error (critical problem) or a
warning (non-critical problem) is encountered.

\param errorHandler a pointer to the error function which
takes the string describing the error in argument.

\param warningHandler a pointer to the warning function which
takes the string describing the error in argument.
*/
void initSwarmBot(void (*errorHandler)(const char *errorString),
  void (*warningHandler)(const char *warningString));

/** Shutdown the s-bot. This function free the low level interfaces to I2C */
void shutdownSwarmBot(void);



/** Set the speed of the tracks.
\param left Left motor speed. [-127;127]
\param right Right motor speed. [-127;127]
*/
void setSpeed(int left, int right);

/** Read the speed of the tracks.
\param left Left motor speed pointer. [-127;127]
\param right Right motor speed pointer. [-127;127]
*/
void getSpeed(int *left, int *right);

/** Read the torque of the tracks.
\param left Left track torque pointer. [-127;127]
\param right Right track torque pointer. [-127;127]
*/
void getTrackTorque(int *left, int *right);



/** Set the rotation angle of the turret.
\param pos Angle of rotation. +/- 0.75 turn. [-100;100]
*/
void setTurretRotation(int pos);

/** Read the rotation angle of the turret.
\return Angle of rotation. +/- 0.75 turn. [-100;100]
*/
int getTurretRotation(void);

/** Read the rotation torque of the turret.
\return pos Torque on turret rotation motor. [-127;127]
*/
int getTurretRotationTorque(void);
```

```
/** Set the elevation of the gripper.
\param elevation Angle of elevation in degree. [-90;90]
*/
void setGripperElevation(int elevation);

/** Read the elevation of the gripper.
\return Angle of elevation in degree. [-90;90]
*/
int getGripperElevation(void);

/** Set the aperture of the gripper.
\param aperture Angle of aperture in degree. [0;40]
*/
void setGripperAperture(int aperture);

/** Read the aperture of the gripper.
\return Angle of aperture of the gripper in degree. [0;40]
*/
int getGripperAperture(void);

/** Read the torque of the gripper.
\return Angler of aperture of the gripper in degree. [-127;127]
*/
int getGripperElevationTorque(void);

/** Read the optical barrier of the gripper.
\param internal Internal barrier pointer. 0 is obstructed, ˜700 is empty. [0;700]
\param external External barrier pointer. 0 is obstructed, ˜700 is empty. [0;700]
\param combined Combined barrier pointer. 0 is obstructed, ˜700 is empty. [0;700]
*/
void getGripperOpticalBarrier(unsigned *internal, unsigned *external,
      unsigned *combined);



/** To be measured */
void setSideArmPos(int elevation, int rotation, int depth);

/** To be measured */
void getSideArmPos(int *elevation, int *rotation, int *depth);

/** To be measured */
void getSideArmTorque(int *elevation, int *rotation, int *depth);

/** Fully open the side-arm’s gripper. */
void setSideArmOpened(void);

/** Fully close the side-arm’s gripper. */
void setSideArmClosed(void);

/** Read the optical barrier of the side arm.
\param internal Internal barrier pointer. 0 is obstructed, ˜700 is empty.[0;700]
\param external External barrier pointer. 0 is obstructed, ˜700 is empty.[0;700]
\param combined Combined barrier pointer. 0 is obstructed, ˜700 is empty.[0;700]
*/
void getSideArmOpticalBarrier(unsigned *internal, unsigned *external,
      unsigned *combined);



/** Read the value of a proximity sensor.
\param sensor Number of sensor. [0;14]
\return Distance to object. 20 is 15 cm, 50 is 5 cm, 1000 is 1 cm, 3700
is 0.5 cm. [0;8191]
*/
unsigned getProximitySensor(unsigned sensor);
```

```
/** Read the value of all proximity sensors.
\param sensors Array of distances to objects. For each value, 20 is 15 cm,
50 is 5 cm, 1000 is 1 cm, 3700 is 0.5 cm. [0;8191]
*/
void getAllProximitySensors(unsigned sensors[15]);

/** Set the sampling rate of the proximity sensors.
\param rate Sampling rate. Can take values SAMPLING_RATE_xx.
*/
void setProximitySamplingRate(SamplingRate rate);



/** Read the value of a ground sensor.
\param sensor Number of sensor. [0;3]
\return Distance to object. 1 is 6mm, 25 is 2mm, 100 is 1mm. [0;255]
*/
unsigned getGroundSensor(unsigned sensor);

/** Read the value of all ground sensors.
\param sensors Array of distances to objects. For each value, 1 is 6mm,
25 is 2mm, 100 is 1mm. [0;255]
*/
void getAllGroundSensors(unsigned sensors[4]);

/** Set the sampling rate of the ground sensors.
\param rate Sampling rate. Can take values SAMPLING_RATE_xx.
*/
void setGroundSamplingRate(SamplingRate rate);



/** Read the value of a light sensor.
\param sensor Number of sensor. [0;7]
\return Intensity. 0 is full dark, ~700 is full light.[0;700]
*/
unsigned getLightSensor(unsigned sensor);

/** Read the value of all light sensors.
\return Array of intensities. For each value, 0 is full dark, ~700 is full
light. [0;700]
*/
void getAllLightSensors(unsigned sensors[8]);

/** Set the color of a light.
\param light Number of light. [0;7]
\param r Red intensity. [0;15]
\param g Green intensity. [0;15]
\param b Blue intensity. [0;15]
\param blink Blinking rate. Can take values BLINKING_RATE_xx.
*/
void setLightColor(unsigned light, unsigned r, unsigned g, unsigned b,
   BlinkingRate blink);

/** Set the color of all lights.
\param r Array of red intensities. For each value [0;15]
\param g Array of green intensities. For each value[0;15]
\param b Array of blue intensities. For each value [0;15]
\param blink Array of blinking rates. For each value, can take values
BLINKING_RATE_xx.
*/
void setAllLightColor(unsigned r[8], unsigned g[8], unsigned b[8],
      BlinkingRate blink[8]);

/** Set the same color for all lights.
\param r Red intensity. [0;15]
\param g Green intensity. [0;15]
```

```
\param b Blue intensity. [0;15]
\param blink Blinking rate. Can take values BLINKING_RATE_xx.
*/
void setSameLightColor(unsigned r, unsigned g, unsigned b, BlinkingRate blink);

/** Set the sampling rate of the light sensors.
\param rate Sampling rate. Can take values SAMPLING_RATE_xx.
*/
void setLightSamplingRate(SamplingRate rate);



/** Read the left temperature and humditiy sensor.
\param temperature Temperature in degree pointer. [-40;123]
\param humidity Relative humidity in % pointer. [0;100]
*/
void getTemperatureAndHumidityLeft(float *temperature, float *humidity);

/** Read the right temperature and humditiy sensor.
\param temperature Temperature in degree pointer. [-40;123]
\param humidity Relative humidity in % pointer. [0;100]
*/
void getTemperatureAndHumidityRight(float *temperature, float *humidity);



/** Read the slope pitch & roll angles of the robot
\param pitch Pitch in degree pointer. [-180;180]
\param roll Roll in degree pointer. [-180;180]
*/
void getSlope(int *pitch, int *roll);

/** Set the sampling rate of the slope sensor.
\param rate Sampling rate. Can take values SAMPLING_RATE_xx.
*/
void setSlopeSamplingRate(SamplingRate rate);



/** Read the current battery voltage
\return Return the current battery voltage 0-10V [0;255]
*/
unsigned getBatteryVoltage(void);


/*
TODO :
- v4l abstraction for camera
*/

/** TO BE DEFINED */
void playMusic(const char *filename);

/** TO BE DEFINED */
void stopMusic(void);


/* Ends C function definitions when using C++ */
#ifdef __cplusplus
}
#endif


#endif
```

Appendix B

Class Diagrams

This appendix presents UML class diagrams of the main modules of the Swarmbot3dsimulator software.

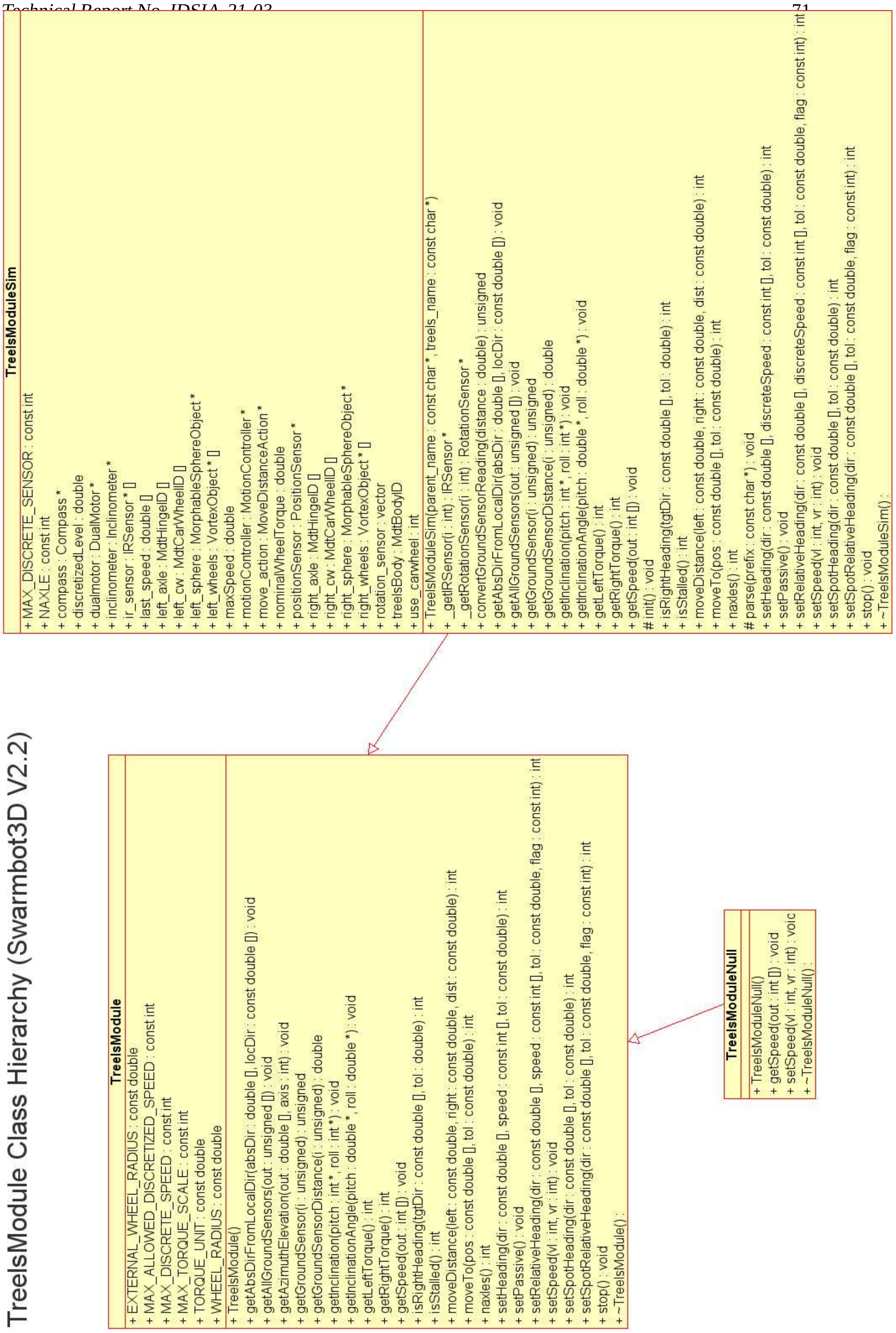

TreelsModule Class Hierarchy (Swarmbot3D V2.2)
TreelsModuleSim
+ MAX_DISCRETE_SENSOR : const int
+ NAXLE : const int
+ compass : Compass *
+ discretizedLevel : double
+ dualmotor : DualMotor *
+ inclinometer : Inclinometer *
+ ir_sensor : IRSensor * []
+ last_speed : double []
+ left_axle : MdtHingeID []
+ left_cw : MdtCarWheelID []
+ left_sphere : MorphableSphereObject *
+ left_wheels : VortexObject * []
+ maxSpeed : double
+ motionController : MotionController *
+ move_action : MoveDistanceAction *
+ nominalWheelTorque : double
+ positionSensor : PositionSensor *
+ right_axle : MdtHingeID []
+ right_cw : MdtCarWheelID []
+ right_sphere : MorphableSphereObject *
+ right_wheels : VortexObject * []
+ rotation_sensor : vector
+ treelsBody : MdtBodyID
+ use_carwheel : int
+ TreelsModuleSim(parent_name : const char *, treels_name : const char *)
+ _getIRSensor(i : int) : IRSensor *
+ _getRotationSensor(i : int) : RotationSensor *
+ convertGroundSensorReading(distance : double) : unsigned
+ getAbsDirFromLocalDir(absDir : double [], locDir : const double []) : void
+ getAllGroundSensors(out : unsigned []) : void
+ getGroundSensor(i : unsigned) : unsigned
+ getGroundSensorDistance(i : unsigned) : double
+ getInclination(pitch : int *, roll : int *) : void
+ getInclinationAngle(pitch : double *, roll : double *) : void
+ getLeftTorque() : int
+ getRightTorque() : int
+ getSpeed(out : int []) : void
# init() : void
+ isRightHeading(tgtDir : const double [], tol : double) : int
+ isStalled() : int
+ moveDistance(left : const double, right : const double, dist : const double) : int
+ moveTo(pos : const double [], tol : const double) : int
+ naxles() : int
# parse(prefix : const char *) : void
+ setHeading(dir : const double [], discreteSpeed : const int [], tol : const double) : int
+ setPassive() : void
+ setRelativeHeading(dir : const double [], discreteSpeed : const int [], tol : const double, flag : const int) : int
+ setSpeed(vl : int, vr : int) : void
+ setSpotHeading(dir : const double [], tol : const double) : int
+ setSpotRelativeHeading(dir : const double [], tol : const double, flag : const int) : int
+ stop() : void
+ ~TreelsModuleSim() :
TreelsModule
+ EXTERNAL_WHEEL_RADIUS : const double
+ MAX_ALLOWED_DISCRETIZED_SPEED : const int
+ MAX_DISCRETE_SPEED : const int
+ MAX_TORQUE_SCALE : const int
+ TORQUE_UNIT : const double
+ WHEEL_RADIUS : const double
+ TreelsModule()
+ getAbsDirFromLocalDir(absDir : double [], locDir : const double []) : void
+ getAllGroundSensors(out : unsigned []) : void
+ getAzimuthElevation(out : double [], axis : int) : void
+ getGroundSensor(i : unsigned) : unsigned
+ getGroundSensorDistance(i : unsigned) : double
+ getInclination(pitch : int *, roll : int *) : void
+ getInclinationAngle(pitch : double *, roll : double *) : void
+ getLeftTorque() : int
+ getRightTorque() : int
+ getSpeed(out : int []) : void
+ isRightHeading(tgtDir : const double [], tol : double) : int
+ isStalled() : int
+ moveDistance(left : const double, right : const double, dist : const double) : int
+ moveTo(pos : const double [], tol : const double) : int
+ naxles() : int
+ setHeading(dir : const double [], speed : const int [], tol : const double) : int
+ setPassive() : void
+ setRelativeHeading(dir : const double [], speed : const int [], tol : const double, flag : const int) : int
+ setSpeed(vl : int, vr : int) : void
+ setSpotHeading(dir : const double [], tol : const double) : int
+ setSpotRelativeHeading(dir : const double [], tol : const double, flag : const int) : int
+ stop() : void
+ ~TreelsModule() :
TreelsModuleNull
+ TreelsModuleNull()
+ getSpeed(out : int []) : void
+ setSpeed(vl : int, vr : int) : voic
+ ~TreelsModuleNull() :

Figure B.1: Class diagram of TreelsModule and related classes.

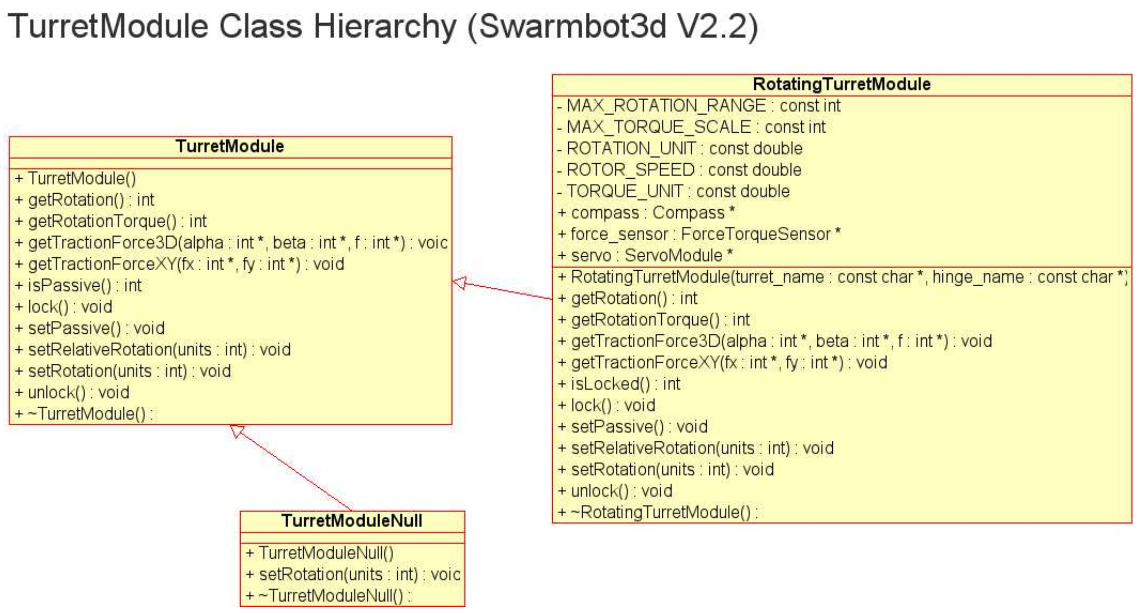

TurretModule Class Hierarchy (Swarmbot3d V2.2)
RotatingTurretModule
- MAX_ROTATION_RANGE : const int
- MAX_TORQUE_SCALE : const int
- ROTATION_UNIT : const double
- ROTOR_SPEED : const double
- TORQUE_UNIT : const double
+ compass : Compass *
+ force_sensor : ForceTorqueSensor *
+ servo : ServoModule *
+ RotatingTurretModule(turret_name : const char *, hinge_name : const char *)
+ getRotation() : int
+ getRotationTorque() : int
+ getTractionForce3D(alpha : int *, beta : int *, f : int *) : void
+ getTractionForceXY(fx : int *, fy : int *) : void
+ isLocked() : int
+ lock() : void
+ setPassive() : void
+ setRelativeRotation(units : int) : void
+ setRotation(units : int) : void
+ unlock() : void
+ ~RotatingTurretModule() :
TurretModule
+ TurretModule()
+ getRotation() : int
+ getRotationTorque() : int
+ getTractionForce3D(alpha : int *, beta : int *, f : int *) : voic
+ getTractionForceXY(fx : int *, fy : int *) : void
+ isPassive() : int
+ lock() : void
+ setPassive() : void
+ setRelativeRotation(units : int) : void
+ setRotation(units : int) : void
+ unlock() : void
+ ~TurretModule() :
TurretModuleNull
+ TurretModuleNull()
+ setRotation(units : int) : voic
+ ~TurretModuleNull() :

Figure B.2: Class diagram of TurretModule and related classes.

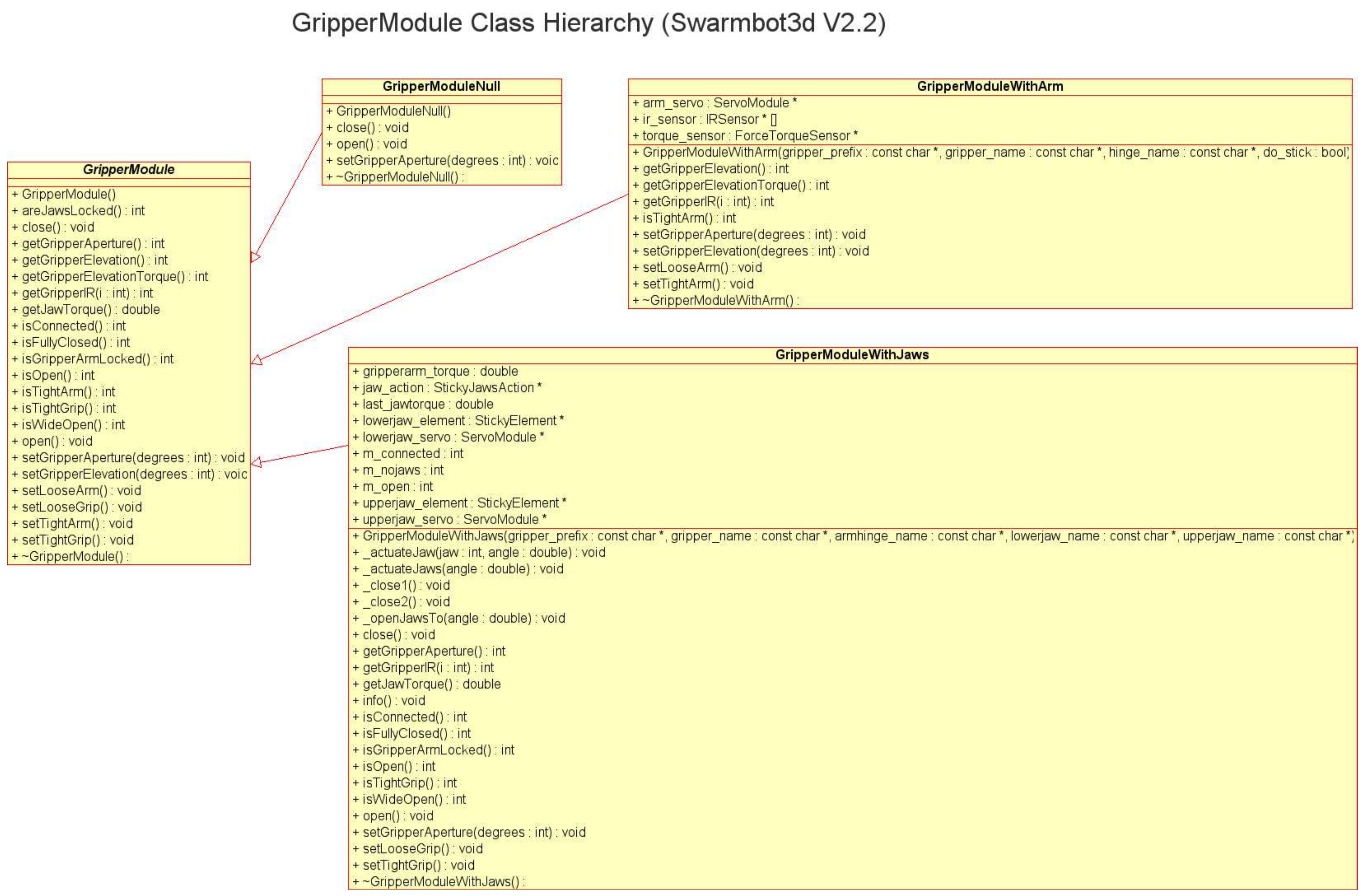

GripperModule Class Hierarchy (Swarmbot3d V2.2)
GripperModule
+ GripperModule()
+ areJawsLocked() : int
+ close() : void
+ getGripperAperture() : int
+ getGripperElevation() : int
+ getGripperElevationTorque() : int
+ getGripperIR(i : int) : int
+ getJawTorque() : double
+ isConnected() : int
+ isFullyClosed() : int
+ isGripperArmLocked() : int
+ isOpen() : int
+ isTightArm() : int
+ isTightGrip() : int
+ isWideOpen() : int
+ open() : void
+ setGripperAperture(degrees : int) : void
+ setGripperElevation(degrees : int) : voic
+ setLooseArm() : void
+ setLooseGrip() : void
+ setTightArm() : void
+ setTightGrip() : void
+ ~GripperModule() :
GripperModuleNull
+ GripperModuleNull()
+ close() : void
+ open() : void
+ setGripperAperture(degrees : int) : voic
+ ~GripperModuleNull() :
GripperModuleWithArm
+ arm_servo : ServoModule *
+ ir_sensor : IRSensor * []
+ torque_sensor : ForceTorqueSensor *
+ GripperModuleWithArm(gripper_prefix : const char *, gripper_name : const char *, hinge_name : const char *, do_stick : bool)
+ getGripperElevation() : int
+ getGripperElevationTorque() : int
+ getGripperIR(i : int) : int
+ isTightArm() : int
+ setGripperAperture(degrees : int) : void
+ setGripperElevation(degrees : int) : void
+ setLooseArm() : void
+ setTightArm() : void
+ ~GripperModuleWithArm() :
GripperModuleWithJaws
+ gripperarm_torque : double
+ jaw_action : StickyJawsAction *
+ last_jawtorque : double
+ lowerjaw_element : StickyElement *
+ lowerjaw_servo : ServoModule *
+ m_connected : int
+ m_nojaws : int
+ m_open : int
+ upperjaw_element : StickyElement *
+ upperjaw_servo : ServoModule *
+ GripperModuleWithJaws(gripper_prefix : const char *, gripper_name : const char *, armhinge_name : const char *, lowerjaw_name : const char *, upperjaw_name : const char *)
+ _actuateJaw(jaw : int, angle : double) : void
+ _actuateJaws(angle : double) : void
+ _close1() : void
+ _close2() : void
+ _openJawsTo(angle : double) : void
+ close() : void
+ getGripperAperture() : int
+ getGripperIR(i : int) : int
+ getJawTorque() : double
+ info() : void
+ isConnected() : int
+ isFullyClosed() : int
+ isGripperArmLocked() : int
+ isOpen() : int
+ isTightGrip() : int
+ isWideOpen() : int
+ open() : void
+ setGripperAperture(degrees : int) : void
+ setLooseGrip() : void
+ setTightGrip() : void
+ ~GripperModuleWithJaws() :

Figure B.3: Class diagram of GripperModule and related classes.
]

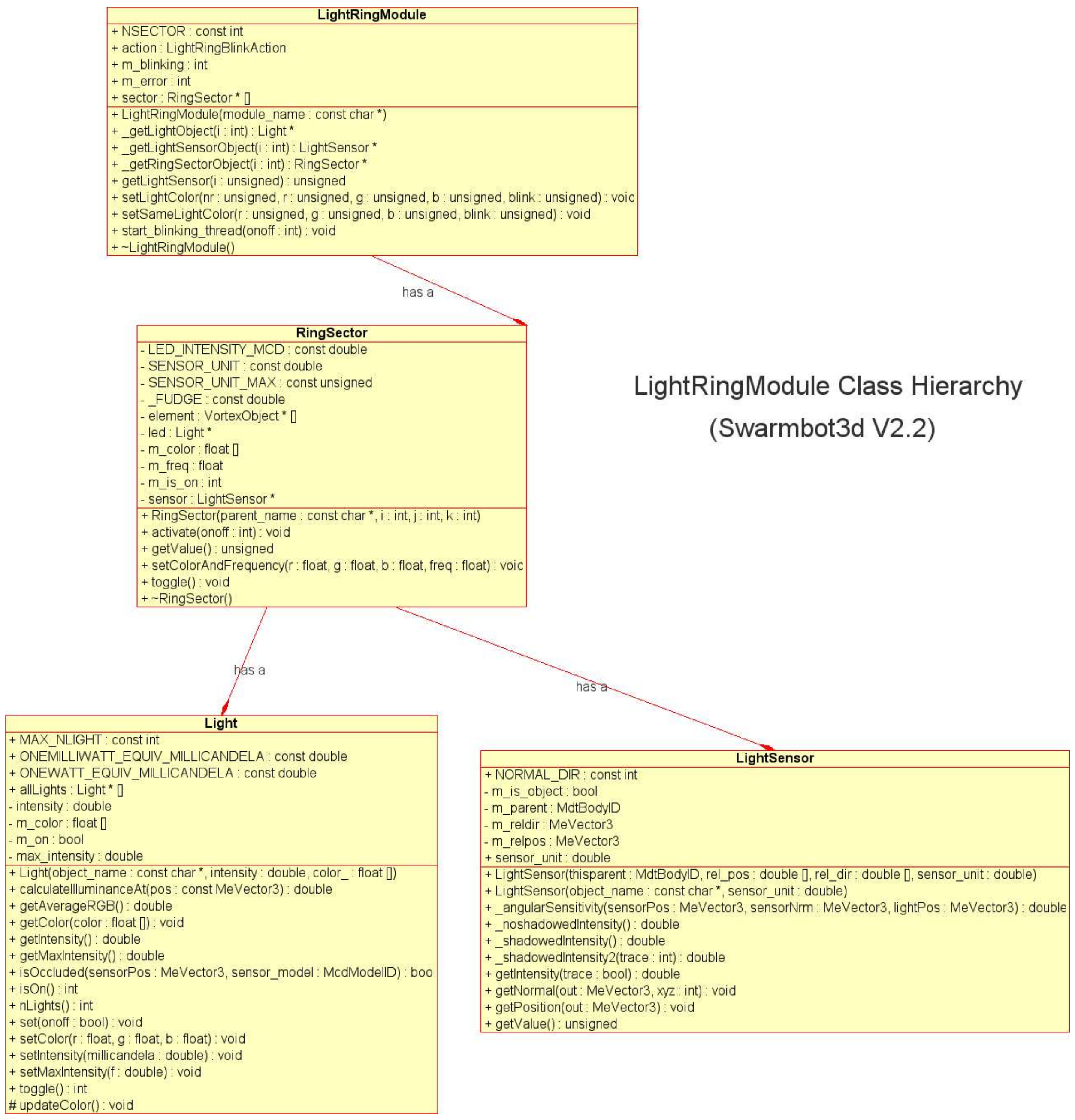

LightRingModule Class Hierarchy
(Swarmbot3d V2.2)
LightRingModule
+ NSECTOR : const int
+ action : LightRingBlinkAction
+ m_blinking : int
+ m_error : int
+ sector : RingSector * []
+ LightRingModule(module_name : const char *)
+ _getLightObject(i : int) : Light *
+ _getLightSensorObject(i : int) : LightSensor *
+ _getRingSectorObject(i : int) : RingSector *
+ getLightSensor(i : unsigned) : unsigned
+ setLightColor(nr : unsigned, r : unsigned, g : unsigned, b : unsigned, blink : unsigned) : voic
+ setSameLightColor(r : unsigned, g : unsigned, b : unsigned, blink : unsigned) : void
+ start_blinking_thread(onoff : int) : void
+ ~LightRingModule()
has a
RingSector
- LED_INTENSITY_MCD : const double
- SENSOR_UNIT : const double
- SENSOR_UNIT_MAX : const unsigned
- _FUDGE : const double
- element : VortexObject * []
- led : Light *
- m_color : float []
- m_freq : float
- m_is_on : int
- sensor : LightSensor *
+ RingSector(parent_name : const char *, i : int, j : int, k : int)
+ activate(onoff : int) : void
+ getValue() : unsigned
+ setColorAndFrequency(r : float, g : float, b : float, freq : float) : voic
+ toggle() : void
+ ~RingSector()
has a
has a
Light
+ MAX_NLIGHT : const int
+ ONEMILLIWATT_EQUIV_MILLICANDELA : const double
+ ONEWATT_EQUIV_MILLICANDELA : const double
+ allLights : Light * []
- intensity : double
- m_color : float []
- m_on : bool
- max_intensity : double
+ Light(object_name : const char *, intensity : double, color_ : float [])
+ calculateIlluminanceAt(pos : const MeVector3) : double
+ getAverageRGB() : double
+ getColor(color : float []) : void
+ getIntensity() : double
+ getMaxIntensity() : double
+ isOccluded(sensorPos : MeVector3, sensor_model : McdModelID) : boo
+ isOn() : int
+ nLights() : int
+ set(onoff : bool) : void
+ setColor(r : float, g : float, b : float) : void
+ setIntensity(millicandela : double) : void
+ setMaxIntensity(f : double) : void
+ toggle() : int
# updateColor() : void
LightSensor
+ NORMAL_DIR : const int
- m_is_object : bool
- m_parent : MdtBodyID
- m_reldir : MeVector3
- m_relpos : MeVector3
+ sensor_unit : double
+ LightSensor(thisparent : MdtBodyID, rel_pos : double [], rel_dir : double [], sensor_unit : double)
+ LightSensor(object_name : const char *, sensor_unit : double)
+ _angularSensitivity(sensorPos : MeVector3, sensorNrm : MeVector3, lightPos : MeVector3) : double
+ _noshadowedIntensity() : double
+ _shadowedIntensity() : double
+ _shadowedIntensity2(trace : int) : double
+ getIntensity(trace : bool) : double
+ getNormal(out : MeVector3, xyz : int) : void
+ getPosition(out : MeVector3) : void
+ getValue() : unsigned

Figure B.4: Class diagram of LightRingModule and related classes.

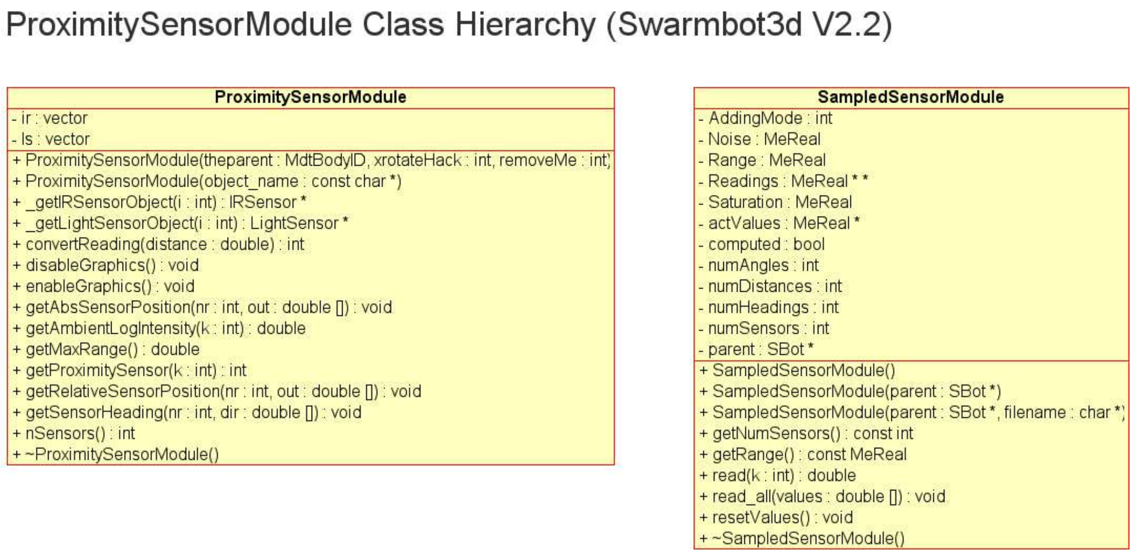

ProximitySensorModule Class Hierarchy (Swarmbot3d V2.2)
ProximitySensorModule
- ir : vector
- ls : vector
+ ProximitySensorModule(theparent : MdtBodyID, xrotateHack : int, removeMe : int)
+ ProximitySensorModule(object_name : const char *)
+ _getIRSensorObject(i : int) : IRSensor *
+ _getLightSensorObject(i : int) : LightSensor *
+ convertReading(distance : double) : int
+ disableGraphics() : void
+ enableGraphics() : void
+ getAbsSensorPosition(nr : int, out : double []) : void
+ getAmbientLogIntensity(k : int) : double
+ getMaxRange() : double
+ getProximitySensor(k : int) : int
+ getRelativeSensorPosition(nr : int, out : double []) : void
+ getSensorHeading(nr : int, dir : double []) : void
+ nSensors() : int
+ ~ProximitySensorModule()
SampledSensorModule
- AddingMode : int
- Noise : MeReal
- Range : MeReal
- Readings : MeReal * *
- Saturation : MeReal
- actValues : MeReal *
- computed : bool
- numAngles : int
- numDistances : int
- numHeadings : int
- numSensors : int
- parent : SBot *
+ SampledSensorModule()
+ SampledSensorModule(parent : SBot *)
+ SampledSensorModule(parent : SBot *, filename : char *)
+ getNumSensors() : const int
+ getRange() : const MeReal
+ read(k : int) : double
+ read_all(values : double []) : void
+ resetValues() : void
+ ~SampledSensorModule()

Figure B.5: Class diagram of ProximitySensorModule and related classes.

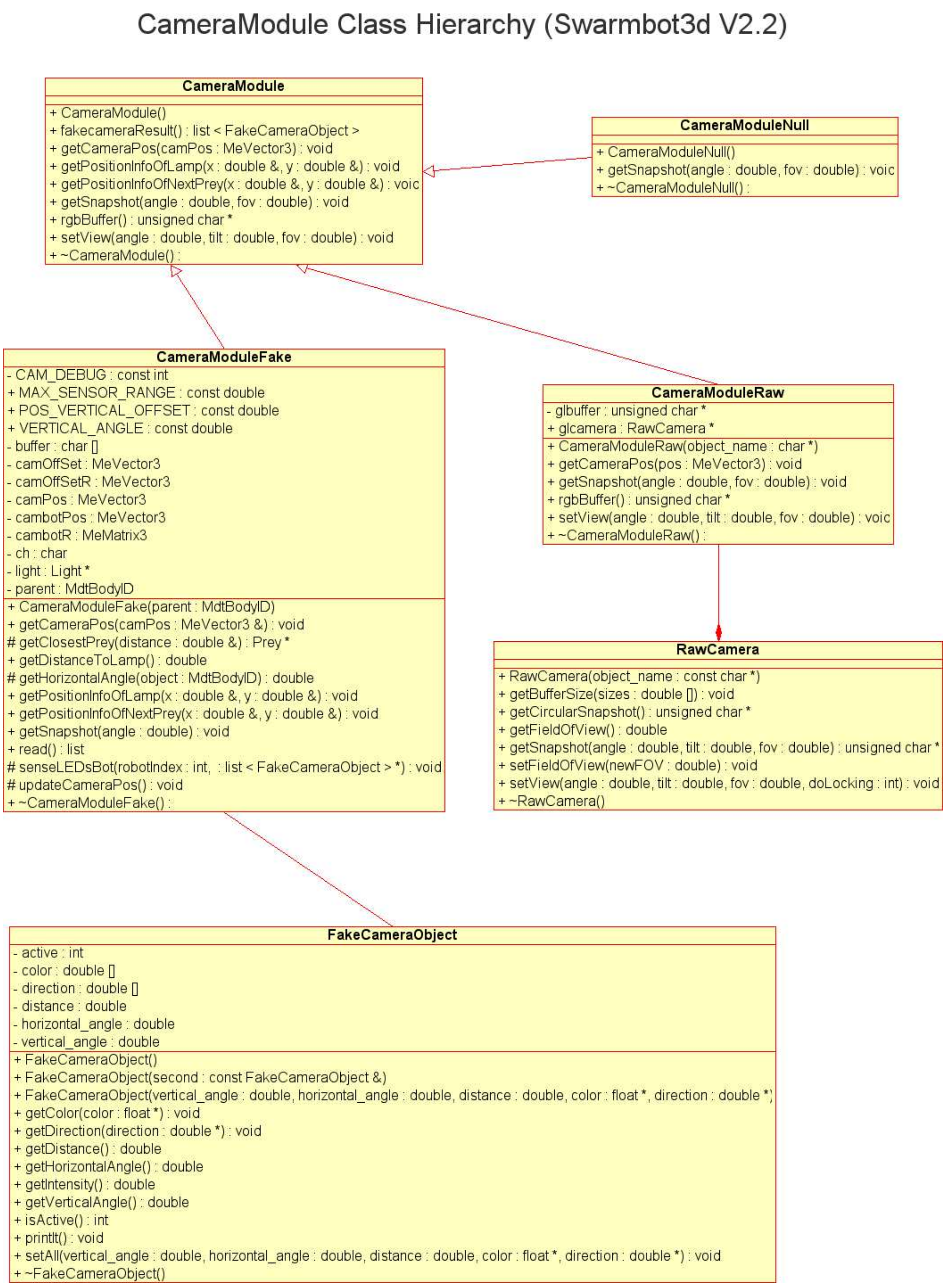

CameraModule Class Hierarchy (Swarmbot3d V2.2)
CameraModule
+ CameraModule()
+ fakecameraResult() : list < FakeCameraObject >
+ getCameraPos(camPos : MeVector3) : void
+ getPositionInfoOfLamp(x : double &, y : double &) : void
+ getPositionInfoOfNextPrey(x : double &, y : double &) : voic
+ getSnapshot(angle : double, fov : double) : void
+ rgbBuffer() : unsigned char *
+ setView(angle : double, tilt : double, fov : double) : void
+ ~CameraModule() :
CameraModuleNull
+ CameraModuleNull()
+ getSnapshot(angle : double, fov : double) : voic
+ ~CameraModuleNull() :
CameraModuleFake
- CAM_DEBUG : const int
+ MAX_SENSOR_RANGE : const double
+ POS_VERTICAL_OFFSET : const double
+ VERTICAL_ANGLE : const double
- buffer : char []
- camOffSet : MeVector3
- camOffSetR : MeVector3
- camPos : MeVector3
- cambotPos : MeVector3
- cambotR : MeMatrix3
- ch : char
- light : Light *
- parent : MdtBodyID
+ CameraModuleFake(parent : MdtBodyID)
+ getCameraPos(camPos : MeVector3 &) : void
# getClosestPrey(distance : double &) : Prey *
+ getDistanceToLamp() : double
# getHorizontalAngle(object : MdtBodyID) : double
+ getPositionInfoOfLamp(x : double &, y : double &) : void
+ getPositionInfoOfNextPrey(x : double &, y : double &) : void
+ getSnapshot(angle : double) : void
+ read() : list
# senseLEDsBot(robotIndex : int, : list < FakeCameraObject > *) : void
# updateCameraPos() : void
+ ~CameraModuleFake() :
CameraModuleRaw
- glbuffer : unsigned char *
+ glcamera : RawCamera *
+ CameraModuleRaw(object_name : char *)
+ getCameraPos(pos : MeVector3) : void
+ getSnapshot(angle : double, fov : double) : void
+ rgbBuffer() : unsigned char *
+ setView(angle : double, tilt : double, fov : double) : voic
+ ~CameraModuleRaw() :
RawCamera
+ RawCamera(object_name : const char *)
+ getBufferSize(sizes : double []) : void
+ getCircularSnapshot() : unsigned char *
+ getFieldOfView() : double
+ getSnapshot(angle : double, tilt : double, fov : double) : unsigned char *
+ setFieldOfView(newFOV : double) : void
+ setView(angle : double, tilt : double, fov : double, doLocking : int) : void
+ ~RawCamera()
FakeCameraObject
- active : int
- color : double []
- direction : double []
- distance : double
- horizontal_angle : double
- vertical_angle : double
+ FakeCameraObject()
+ FakeCameraObject(second : const FakeCameraObject &)
+ FakeCameraObject(vertical_angle : double, horizontal_angle : double, distance : double, color : float *, direction : double *)
+ getColor(color : float *) : void
+ getDirection(direction : double *) : void
+ getDistance() : double
+ getHorizontalAngle() : double
+ getIntensity() : double
+ getVerticalAngle() : double
+ isActive() : int
+ printIt() : void
+ setAll(vertical_angle : double, horizontal_angle : double, distance : double, color : float *, direction : double *) : void
+ ~FakeCameraObject()

Figure B.6: Class diagram of CameraModule and related classes.

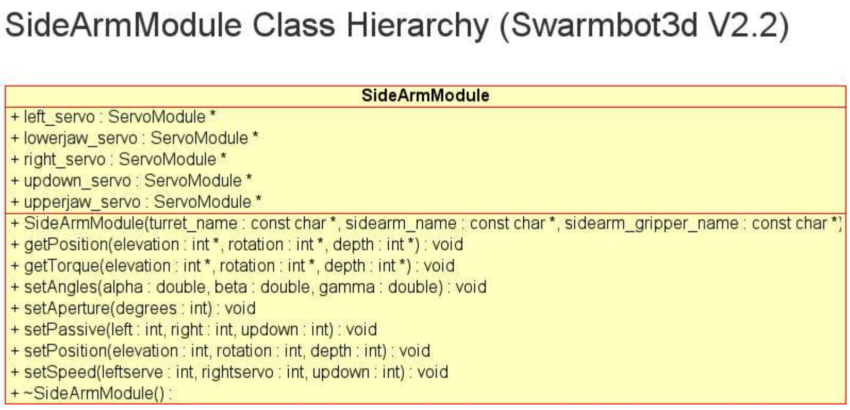

SideArmModule Class Hierarchy (Swarmbot3d V2.2)
SideArmModule
+ left_servo : ServoModule *
+ lowerjaw_servo : ServoModule *
+ right_servo : ServoModule *
+ updown_servo : ServoModule *
+ upperjaw_servo : ServoModule *
+ SideArmModule(turret_name : const char *, sidearm_name : const char *, sidearm_gripper_name : const char *)
+ getPosition(elevation : int *, rotation : int *, depth : int *) : void
+ getTorque(elevation : int *, rotation : int *, depth : int *) : void
+ setAngles(alpha : double, beta : double, gamma : double) : void
+ setAperture(degrees : int) : void
+ setPassive(left : int, right : int, updown : int) : void
+ setPosition(elevation : int, rotation : int, depth : int) : void
+ setSpeed(leftserve : int, rightservo : int, updown : int) : void
+ ~SideArmModule() :

Figure B.7: Class diagram of SideArmModule and related classes.

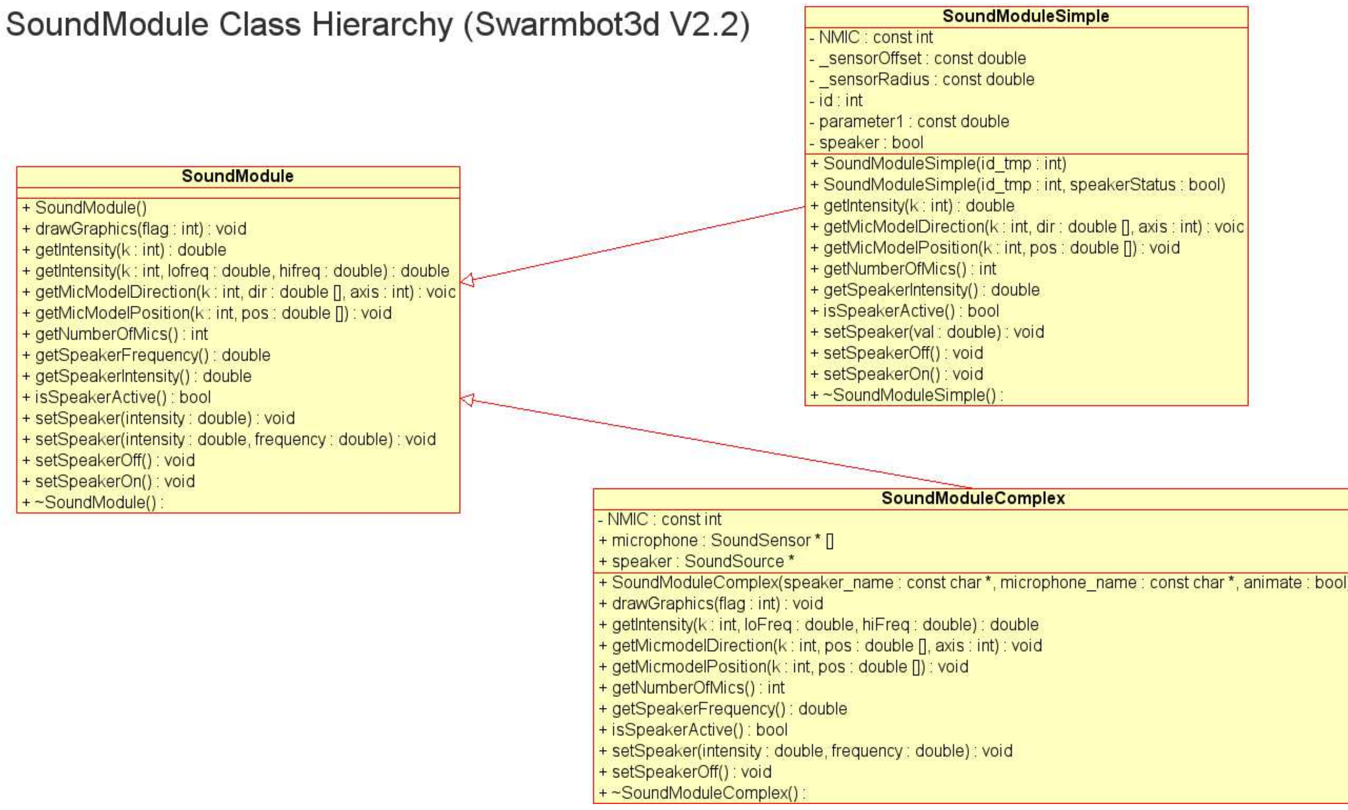

SoundModule Class Hierarchy (Swarmbot3d V2.2)
SoundModule
+ SoundModule()
+ drawGraphics(flag : int) : void
+ getIntensity(k : int) : double
+ getIntensity(k : int, lofreq : double, hifreq : double) : double
+ getMicModelDirection(k : int, dir : double [], axis : int) : voic
+ getMicModelPosition(k : int, pos : double []) : void
+ getNumberOfMics() : int
+ getSpeakerFrequency() : double
+ getSpeakerIntensity() : double
+ isSpeakerActive() : bool
+ setSpeaker(intensity : double) : void
+ setSpeaker(intensity : double, frequency : double) : void
+ setSpeakerOff() : void
+ setSpeakerOn() : void
+ ~SoundModule() :
SoundModuleSimple
- NMIC : const int
- _sensorOffset : const double
- _sensorRadius : const double
- id : int
- parameter1 : const double
- speaker : bool
+ SoundModuleSimple(id_tmp : int)
+ SoundModuleSimple(id_tmp : int, speakerStatus : bool)
+ getIntensity(k : int) : double
+ getMicModelDirection(k : int, dir : double [], axis : int) : voic
+ getMicModelPosition(k : int, pos : double []) : void
+ getNumberOfMics() : int
+ getSpeakerIntensity() : double
+ isSpeakerActive() : bool
+ setSpeaker(val : double) : void
+ setSpeakerOff() : void
+ setSpeakerOn() : void
+ ~SoundModuleSimple() :
SoundModuleComplex
- NMIC : const int
+ microphone : SoundSensor * []
+ speaker : SoundSource *
+ SoundModuleComplex(speaker_name : const char *, microphone_name : const char *, animate : bool
+ drawGraphics(flag : int) : void
+ getIntensity(k : int, loFreq : double, hiFreq : double) : double
+ getMicmodelDirection(k : int, pos : double [], axis : int) : void
+ getMicmodelPosition(k : int, pos : double []) : void
+ getNumberOfMics() : int
+ getSpeakerFrequency() : double
+ isSpeakerActive() : bool
+ setSpeaker(intensity : double, frequency : double) : void
+ setSpeakerOff() : void
+ ~SoundModuleComplex() :

Figure B.8: Class diagram of SoundModule and related classes.

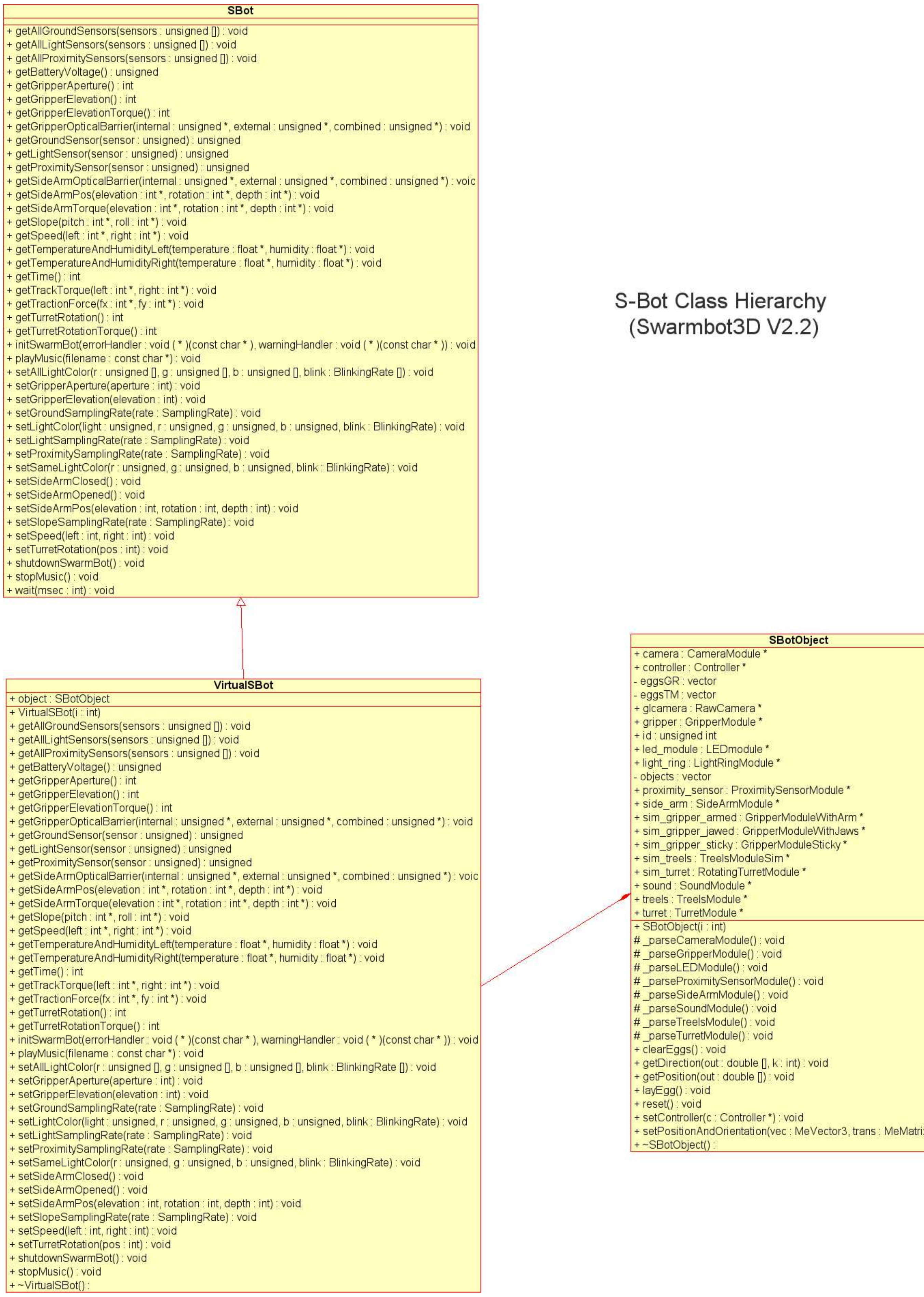

S-Bot Class Hierarchy
(Swarmbot3D V2.2)
SBot
+ getAllGroundSensors(sensors : unsigned []) : void
+ getAllLightSensors(sensors : unsigned []) : void
+ getAllProximitySensors(sensors : unsigned []) : void
+ getBatteryVoltage() : unsigned
+ getGripperAperture() : int
+ getGripperElevation() : int
+ getGripperElevationTorque() : int
+ getGripperOpticalBarrier(internal : unsigned *, external : unsigned *, combined : unsigned *) : void
+ getGroundSensor(sensor : unsigned) : unsigned
+ getLightSensor(sensor : unsigned) : unsigned
+ getProximitySensor(sensor : unsigned) : unsigned
+ getSideArmOpticalBarrier(internal : unsigned *, external : unsigned *, combined : unsigned *) : voic
+ getSideArmPos(elevation : int *, rotation : int *, depth : int *) : void
+ getSideArmTorque(elevation : int *, rotation : int *, depth : int *) : void
+ getSlope(pitch : int *, roll : int *) : void
+ getSpeed(left : int *, right : int *) : void
+ getTemperatureAndHumidityLeft(temperature : float *, humidity : float *) : void
+ getTemperatureAndHumidityRight(temperature : float *, humidity : float *) : void
+ getTime() : int
+ getTrackTorque(left : int *, right : int *) : void
+ getTractionForce(fx : int *, fy : int *) : void
+ getTurretRotation() : int
+ getTurretRotationTorque() : int
+ initSwarmBot(errorHandler : void (*)(const char *), warningHandler : void (*)(const char *)) : void
+ playMusic(filename : const char *) : void
+ setAllLightColor(r : unsigned [], g : unsigned [], b : unsigned [], blink : BlinkingRate []) : void
+ setGripperAperture(aperture : int) : void
+ setGripperElevation(elevation : int) : void
+ setGroundSamplingRate(rate : SamplingRate) : void
+ setLightColor(light : unsigned, r : unsigned, g : unsigned, b : unsigned, blink : BlinkingRate) : void
+ setLightSamplingRate(rate : SamplingRate) : void
+ setProximitySamplingRate(rate : SamplingRate) : void
+ setSameLightColor(r : unsigned, g : unsigned, b : unsigned, blink : BlinkingRate) : void
+ setSideArmClosed() : void
+ setSideArmOpened() : void
+ setSideArmPos(elevation : int, rotation : int, depth : int) : void
+ setSlopeSamplingRate(rate : SamplingRate) : void
+ setSpeed(left : int, right : int) : void
+ setTurretRotation(pos : int) : void
+ shutdownSwarmBot() : void
+ stopMusic() : void
+ wait(msec : int) : void
VirtualSBot
+ object : SBotObject
+ VirtualSBot(i : int)
+ getAllGroundSensors(sensors : unsigned []) : void
+ getAllLightSensors(sensors : unsigned []) : void
+ getAllProximitySensors(sensors : unsigned []) : void
+ getBatteryVoltage() : unsigned
+ getGripperAperture() : int
+ getGripperElevation() : int
+ getGripperElevationTorque() : int
+ getGripperOpticalBarrier(internal : unsigned *, external : unsigned *, combined : unsigned *) : void
+ getGroundSensor(sensor : unsigned) : unsigned
+ getLightSensor(sensor : unsigned) : unsigned
+ getProximitySensor(sensor : unsigned) : unsigned
+ getSideArmOpticalBarrier(internal : unsigned *, external : unsigned *, combined : unsigned *) : voic
+ getSideArmPos(elevation : int *, rotation : int *, depth : int *) : void
+ getSideArmTorque(elevation : int *, rotation : int *, depth : int *) : void
+ getSlope(pitch : int *, roll : int *) : void
+ getSpeed(left : int *, right : int *) : void
+ getTemperatureAndHumidityLeft(temperature : float *, humidity : float *) : void
+ getTemperatureAndHumidityRight(temperature : float *, humidity : float *) : void
+ getTime() : int
+ getTrackTorque(left : int *, right : int *) : void
+ getTractionForce(fx : int *, fy : int *) : void
+ getTurretRotation() : int
+ getTurretRotationTorque() : int
+ initSwarmBot(errorHandler : void (*)(const char *), warningHandler : void (*)(const char *)) : void
+ playMusic(filename : const char *) : void
+ setAllLightColor(r : unsigned [], g : unsigned [], b : unsigned [], blink : BlinkingRate []) : void
+ setGripperAperture(aperture : int) : void
+ setGripperElevation(elevation : int) : void
+ setGroundSamplingRate(rate : SamplingRate) : void
+ setLightColor(light : unsigned, r : unsigned, g : unsigned, b : unsigned, blink : BlinkingRate) : void
+ setLightSamplingRate(rate : SamplingRate) : void
+ setProximitySamplingRate(rate : SamplingRate) : void
+ setSameLightColor(r : unsigned, g : unsigned, b : unsigned, blink : BlinkingRate) : void
+ setSideArmClosed() : void
+ setSideArmOpened() : void
+ setSideArmPos(elevation : int, rotation : int, depth : int) : void
+ setSlopeSamplingRate(rate : SamplingRate) : void
+ setSpeed(left : int, right : int) : void
+ setTurretRotation(pos : int) : void
+ shutdownSwarmBot() : void
+ stopMusic() : void
+ ~VirtualSBot() :
SBotObject
+ camera : CameraModule *
+ controller : Controller *
- eggsGR : vector
- eggsTM : vector
+ glcamera : RawCamera *
+ gripper : GripperModule *
+ id : unsigned int
+ led_module : LEDmodule *
+ light_ring : LightRingModule *
- objects : vector
+ proximity_sensor : ProximitySensorModule *
+ side_arm : SideArmModule *
+ sim_gripper_armed : GripperModuleWithArm *
+ sim_gripper_jawed : GripperModuleWithJaws *
+ sim_gripper_sticky : GripperModuleSticky *
+ sim_treels : TreelsModuleSim *
+ sim_turret : RotatingTurretModule *
+ sound : SoundModule *
+ treels : TreelsModule *
+ turret : TurretModule *
+ SBotObject(i : int)
# _parseCameraModule() : void
# _parseGripperModule() : void
# _parseLEDModule() : void
# _parseProximitySensorModule() : void
# _parseSideArmModule() : void
# _parseSoundModule() : void
# _parseTreelsModule() : void
# _parseTurretModule() : void
+ clearEggs() : void
+ getDirection(out : double [], k : int) : void
+ getPosition(out : double []) : void
+ layEgg() : void
+ reset() : void
+ setController(c : Controller *) : void
+ setPositionAndOrientation(vec : MeVector3, trans : MeMatri
+ ~SBotObject() :

Figure B.9: Class diagram of SBot and related classes.

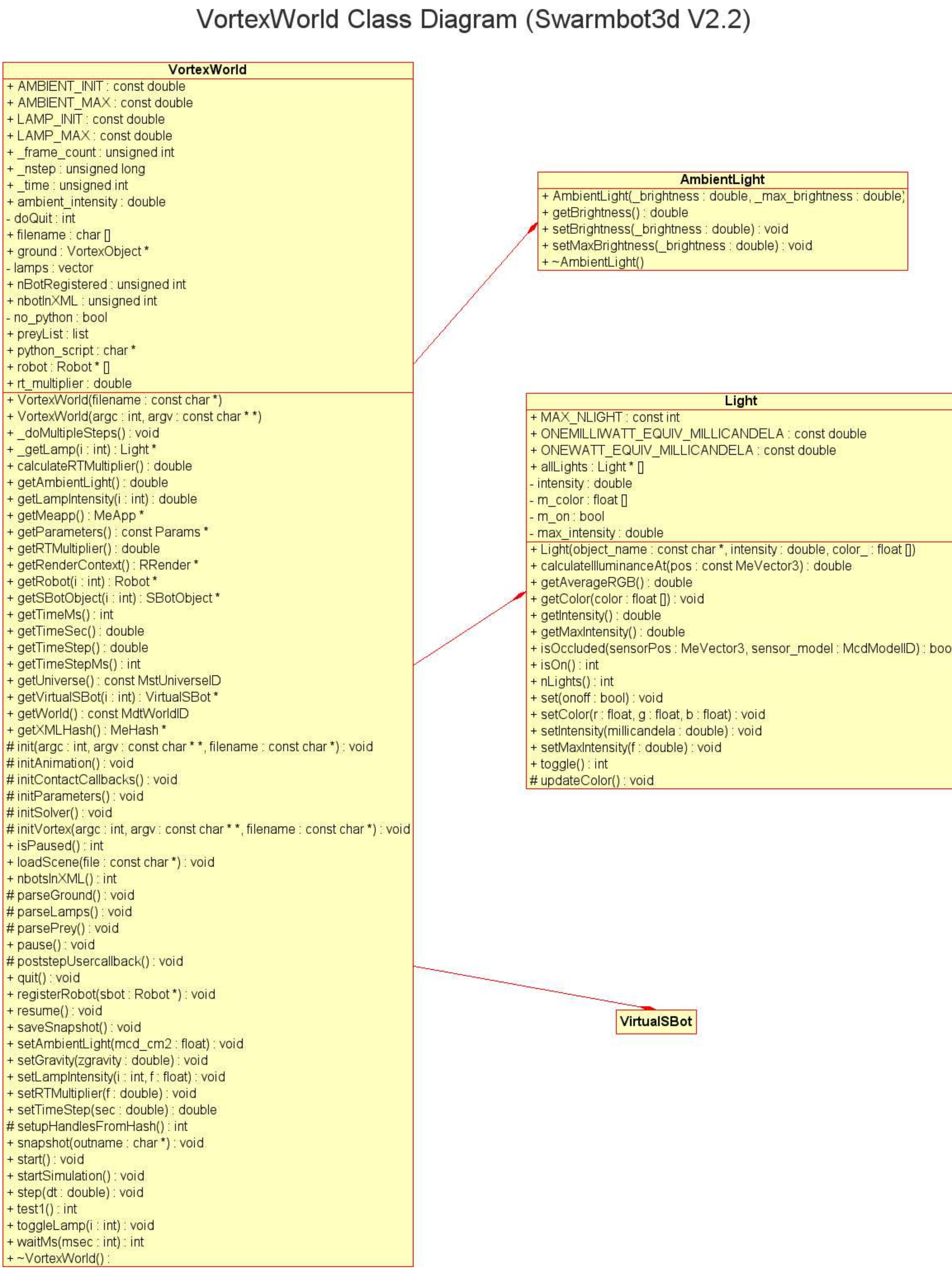

VortexWorld Class Diagram (Swarmbot3d V2.2)
VortexWorld
+ AMBIENT_INIT : const double
+ AMBIENT_MAX : const double
+ LAMP_INIT : const double
+ LAMP_MAX : const double
+ _frame_count : unsigned int
+ _nstep : unsigned long
+ _time : unsigned int
+ ambient_intensity : double
- doQuit : int
+ filename : char []
+ ground : VortexObject *
- lamps : vector
+ nBotRegistered : unsigned int
+ nbotInXML : unsigned int
- no_python : bool
+ preyList : list
+ python_script : char *
+ robot : Robot * []
+ rt_multiplier : double
+ VortexWorld(filename : const char *)
+ VortexWorld(argc : int, argv : const char * *)
+ _doMultipleSteps() : void
+ _getLamp(i : int) : Light *
+ calculateRTMultiplier() : double
+ getAmbientLight() : double
+ getLampIntensity(i : int) : double
+ getMeapp() : MeApp *
+ getParameters() : const Params *
+ getRTMultiplier() : double
+ getRenderContext() : RRender *
+ getRobot(i : int) : Robot *
+ getSBotObject(i : int) : SBotObject *
+ getTimeMs() : int
+ getTimeSec() : double
+ getTimeStep() : double
+ getTimeStepMs() : int
+ getUniverse() : const MstUniverseID
+ getVirtualSBot(i : int) : VirtualSBot *
+ getWorld() : const MdtWorldID
+ getXMLHash() : MeHash *
# init(argc : int, argv : const char * *, filename : const char *) : void
# initAnimation() : void
# initContactCallbacks() : void
# initParameters() : void
# initSolver() : void
# initVortex(argc : int, argv : const char * *, filename : const char *) : void
+ isPaused() : int
+ loadScene(file : const char *) : void
+ nbotsInXML() : int
# parseGround() : void
# parseLamps() : void
# parsePrey() : void
+ pause() : void
# poststepUsercallback() : void
+ quit() : void
+ registerRobot(sbot : Robot *) : void
+ resume() : void
+ saveSnapshot() : void
+ setAmbientLight(mcd_cm2 : float) : void
+ setGravity(zgravity : double) : void
+ setLampIntensity(i : int, f : float) : void
+ setRTMultiplier(f : double) : void
+ setTimeStep(sec : double) : double
# setupHandlesFromHash() : int
+ snapshot(outname : char *) : void
+ start() : void
+ startSimulation() : void
+ step(dt : double) : void
+ test1() : int
+ toggleLamp(i : int) : void
+ waitMs(msec : int) : int
+ ~VortexWorld() :
AmbientLight
+ AmbientLight(_brightness : double, _max_brightness : double)
+ getBrightness() : double
+ setBrightness(_brightness : double) : void
+ setMaxBrightness(_brightness : double) : void
+ ~AmbientLight()
Light
+ MAX_NLIGHT : const int
+ ONEMILLIWATT_EQUIV_MILLICANDELA : const double
+ ONEWATT_EQUIV_MILLICANDELA : const double
+ allLights : Light * []
- intensity : double
- m_color : float []
- m_on : bool
- max_intensity : double
+ Light(object_name : const char *, intensity : double, color_ : float [])
+ calculateIlluminanceAt(pos : const MeVector3) : double
+ getAverageRGB() : double
+ getColor(color : float []) : void
+ getIntensity() : double
+ getMaxIntensity() : double
+ isOccluded(sensorPos : MeVector3, sensor_model : McdModelID) : boo
+ isOn() : int
+ nLights() : int
+ set(onoff : bool) : void
+ setColor(r : float, g : float, b : float) : void
+ setIntensity(millicandela : double) : void
+ setMaxIntensity(f : double) : void
+ toggle() : int
# updateColor() : void
VirtualSBot

Figure B.10: Class diagram of VortexWorld and related classes.