Simulation 3D d'un groupe de robots

by

Tran-Dinh Dũng David

Année académique 2004-2005
Université Libre de Bruxelles, Faculté de Sciences Appliquées
Avenue Franklin Roosevelt 50, 1050 Brussels, Belgium
david.trandinh@gmail.com

Supervised by

Prof. Marco Dorigo

Directeur de Recherches du FNRS
Université Libre de Bruxelles, IRIDIA
Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium
mdorigo@ulb.ac.be

MEMOIRE DE FIN D'ETUDES
PRESENTE PAR Tran-Dinh Dũng David
EN VUE DE L'OBTENTION DU GRADE
D'INGENIEUR CIVIL INFORMATICIEN

Contents

Abstract

In this work, we present a 3D simulator designed for collective robotics and particularly for the swarm-bots. The Swarm-bot is an artifact composed of a swarm of s-bots, mobile robots with the ability to connect to/disconnect from each other. Challenges associated with such a simulator include complex dynamics and sensors simulation. The physics engine chosen is ODE, which proved to be fast and accurate enough. Additionally, important requirements are a graphical user interface, as well as a configurability via XML, and multiplatform support. We study the actual robot's actuators and sensors, and explain how they can be modeled. Finally we present the results of a series of benchmarks and experiments involving simple controllers, such as obstacle avoidance and phototaxis.

Chapter 1

Introduction

The aim of this work is the design and implementation of a simulator for a group of robots. The main objectives are fast and accurate simulations, an easy to use graphical interface, high configurability via XML, modular programming, multiplatform compatibility, and a user documentation.

Simulations are powerful tools 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 spreedsheets or flowcharts[Diamond, 2002]. In the field of robotics, simulations are especially useful to engineers to design and test controllers avoiding the risk of damaging the robot. Moreover, simulations are useful to design controllers through *artificial evolution*, a population-based metaheuristic optimization algorithm that uses mechanisms inspired by biological evolution, such as reproduction, mutation and recombination [5]. Simulations also allow to study systems comprised of more robots than actually available. This is especially true with collective robotics, in which a high number of robots could be required. Another utility of simulations is the testing and validation of ideas for new hardware features.

As mentioned above, this work is tailored particularly for a collective robotic system, and more specifically for the swarm-bots, that is, a robotic artifact composed of a number of simple, autonomous robots called s-bots, able to connect to/disconnect from each other. *Swarm robotics* is a novel approach to the design and implementation of robotic systems, inspired by studies in swarm intelligence [2]. These systems are composed of *swarms* of robots which tightly interact and cooperate to reach their goal. Swarm robotics can be considered as an instance of the more general field of collective robotics. It is inspired by the social insect metaphor and emphasizes

aspects like decentralization of the control, limited communication abilities among robots, emergence of global behavior and robustness. In a swarm robotic system, although each single robot composing the swarm is a fully autonomous robot, the swarm as a whole can solve problems that the single robot can not solve because of physical constraints or limited abilities. We will examine the swarm-bot in detail in Section 1.1.

Many challenges are associated with the development of a simulator for a group of robots. The most relevant ones are related to the simulation of a system presenting *complex dynamics*. Robots can interact with the environment but also with each other. Particularly, in a swarm-bot, s-bots can connect to each other, thus creating forces at the joints that must be taken into account when simulating such a system. Such a complexity might make the simulation unstable or too slow for a proper usage. Therefore, the choice of the physics library is very important and validation tests need to be run. Another challenge is given by *sensors modeling*. The goal is to reproduce in simulation the same perception the real robot has. This task is non trivial, especially considering that in a multirobot domain, the environmental situation continuously changes. A final issue is given by the software design: the main goal is keeping the complexity as low as possible, in order to obtain fast and reliable simulations.

We faced the above challenges while developing the software. All the required sensing techniques have been implemented and validated through several tests taking into account both accuracy and speed. An intuitive user interface has been developed, featuring 3D rendering with special effects as well as a console for easy debugging, and easy user-defined mapping of mouse and keyboard controls. Additionally, the command line controls allow powerful scripting. Configurability is ensured through XML descriptions. It comprises models creation and parameters for the sensors and controllers.

The simulator has been programmed in C++, it only relies on open source libraries such as SDL and OpenGL, and can be compiled on linux and windows with the same results. Various benchmark tests have been performed. Finally, four examples of controllers for the robots have been provided, they consist in obstacle avoidance, phototaxis [1], assembly and motion on rough terrain.

In the following section, we will briefly describe the swarm-bot (see Section 1.1) and the implementation choices we made (see Section 1.2). Finally

[1]Phototaxis occurs when a whole organism moves in response to the stimulus light.

Section 1.3 details the layout of the report.

1.1 The swarm-bot

A swarm-bot is defined as an artifact composed of a swarm of s-bots, mobile robots with the ability to connect to/disconnect from each other. S-bots have simple sensors and motors and limited computational capabilities [4,6]. Their physical links are used to assemble into a swarm-bot able to solve problems that can not be solved by a single s-bot (see Figure 1.1).

Figure 1.1: Graphical visualization of an s-bot

In the swarm-bot form, the s-bots are attached to each other and the robotic system is a single whole that can move and reconfigure along the way when needed. For example, it might have to adopt different shapes in order to go through a narrow passage or overcome an obstacle.

Physical connections between s-bots are important for building pulling chains, as for example in an object retrieval scenario (see Figure 1.2a). They can also serve as support if the swarm-bot is going over a hole larger than a single s-bot, as exemplified in Figure 1.2b, or when the swarm-bot is passing through a steep concave region, in a navigation on rough terrain scenario.

Anyway, there might be occasions in which a swarm of independent s-bots is more efficient: for example, when searching for a goal location or when tracking an optimal path to a goal.

The above examples represent the family of tasks a swarm-bot should be able to perform.

The hardware design of an s-bot is particularly innovative, as far as it concerns both its actuators and its sensing devices. The s-bot is equipped with an innovative traction system which makes use of both tracks and wheels as

(a) (b)

Figure 1.2: Graphical visualization of possible scenarios involving a swarm-bot. (a) Retrieving a circular object. (b) Passing over a trough.

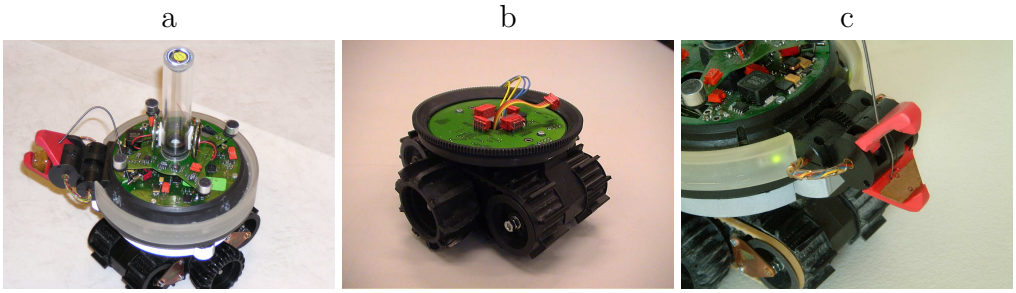


Figure 1.3: (a) A s-bot; (b) the traction system of a s-bot; (c) the s-bot's gripper.

illustrated in figure 1.3b. The wheel and the track on a same side are driven by the same motor, building a differential drive system controlled by two motors. This combination of tracks and wheels is labelled *Differential Treels* © *Drive*.[2] Such a combination has two advantages. First, it allows an efficient rotation on the spot due to the larger diameter and position of the wheels. Second, it gives to the traction system a shape close to the cylindrical one of the main body (turret), avoiding in this way the typical rectangular shape of simple tracks and thus improving the s-bot mobility.

The s-bot's traction system can rotate with respect to the main body by means of a motorised joint. The turret holds a gripper for establishing rigid connections between two s-bots or between an s-bot and an object (see figure 1.3a). The gripper is mounted on a horizontal active axis which, if necessary, can be used to lift another s-bot. Such a gripper has a very large

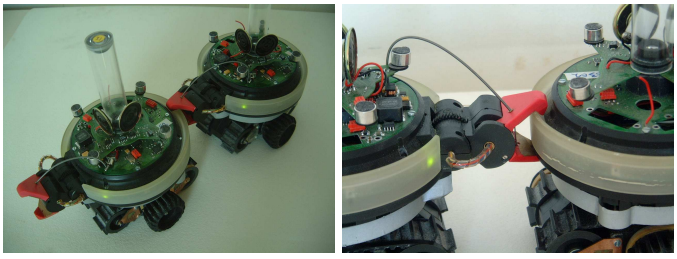
[2]Treels is a contraction of TRacks and whEELS

acceptance area allowing it to realize a secure grasp at a wide angle range. The s-bot gripper can grasp another s-bot on a T-shaped ring placed around the s-bot turret (see Figure 1.4b. If it is not completely closed, such a grasp allow some relative displacement of the two joined robots. If the grasp is firm, the gripper ensures a very rigid connection which can even sustain the lifting up of another s-bot.

An s-bot is provided with many sensory systems, useful for the perception of the surrounding environment or for proprioception. Infrared proximity sensors are distributed around the rotating turret, and can be used for detection of obstacles and other s-bots. Four proximity sensors are placed under the chassis, and can be used for perceiving holes or the terrain's roughness. Additionally, an s-bot is provided with eight light sensors, two temperature/humidity sensors, a 3-axes accelerometer and incremental encoders on each degree of freedom.

Each robot is also equipped with sensors and devices to detect and communicate with other s-bots, such as an omni-directional camera, colored LEDs around the robot's turret, microphones and loudspeakers. Eight groups of three colored LEDs each—red, green and blue—are mounted around the turret, and they can be used to emit a color that represents a particular internal state of the robot. The color emitted by a robot can be detected using the omni-directional camera, which allow to grab panoramic views of the scene surrounding an s-bot.

In addition to a large number of sensors for perceiving the environment, several sensors provide each s-bot with information about physical contacts, efforts, and reactions at the interconnection joints with other s-bots. These include torque sensors on most joints as well as a traction sensor to measure



(a) (b)

Figure 1.4: (a) Two connected s-bots; (b) detailed view of a connection between two s-bots

the pulling/pushing forces exerted on the s-bot's turret. The traction sensor is placed at the junction between the turret and the chassis. This sensor measures the direction (i.e., the angle with respect to the chassis orientation) and the intensity of the force of traction (henceforth called "traction") that the turret exerts on the chassis. The traction perceived by one robot can be caused either by the force applied by the robot itself while pulling/pushing an object grasped through the gripper element, or by the mismatch of its movement with respect to the movement of other robots connected to it, or by both the previous circumstances at the same time. The turret of a s-bot physically integrates through a vector summation the forces that are applied to it by another s-bot, as well as the force the s-bot itself applies to a grasped object [1].

1.2 Simulator taxonomy

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

Static vs. Dynamic simulation Static simulations 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 behavior of a system are usually lost. Dynamic simulations compute the future effect of a parameter into their model by iteratively evolving the model through time. They can be classified into three categories.

Continuous, Discrete and Hybrid simulations 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 occurence of a series of discrete events. Because of this, it is possible to distinguish three possible ways of letting time evolve: continously, discretely, or hybridly between the previous two cases.

- Continuous simulation. The values of the 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.

- 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 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.

- Hybrid simulation. This type joins some characteristics of the previous 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 ar dependent on the time taken to carry them out.

Kinematics vs. Dynamics simulation Continuous simulations can be divided depending on the type of physics employed. As mentioned earlier, since continuous simulations evolve their simulated physical world, they need to define mathematicals 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.

- Kinematic: each object is treated as a massless pointlike particle and its velocity is modified directly

- Dynamic: each object is defined by a mass and a geometry, its velocity is modified indirectly by the application of forces

We are interested to see the evolution of a group of robots over time, for that reason a continous simulation is required. If we were only interested in the motion of the robots, a kinematic model would be sufficient, but robots are able to grip each other, are able to lift objects, this will create forces that we can not disregard. Additionally, thse forces can be utilized wisely in controllers. For these reasons we made the choice of using a dynamic simulation.

1.3 Layout of the report

In chapter 2, we describe the simulation techniques we used. We first motivate the choice of the physics engine used and its features. Then, we discuss how the robots can be simulated correctly. In chapter 3, we examine the simulation model in detail, namely how to model the robot's actuators and sensors. Chapter 4 covers the software design, describing the UML class diagram and the usage of XML for the simulation. In chapter 5, we comment on the results obtained from benchmarking tests on the simulator under various conditions, to better understand its features and limits. Appendix A contains the simulator documentation.

Chapter 2

Simulation techniques

2.1 Choice of the Physics engine

A physics engine is a computer program that simulates the dynamics of multiple objects taking into account variables such as mass, velocity, friction and wind resistance. A physics engine can therefore closely approximate what happens in the real world.

There are generally two types of physics engines: real-time and high precision. A high precision physics engine requires a long computing time to calculate precise physics of a given world. For this reason it is used in situations that do not require real-time such as scientific simulations or some computer animated movies. In video games, or other forms of interactive computing, the physics engine needs to simplify its calculations and their accuracy so that they can be performed in time. This is referred to as real-time physics.

In the following we review the most commonly used physics engines. Required features are speed, support of necessary geometries and joints including limits, a collision detection engine, and multiplatform C++ API [1].

DynaMechs http://dynamechs.sourceforge.net/

This is a multibody dynamic simulation library, it was originally developed for two graduate research projects requiring a real-time dynamic and hydrodynamic simulation. The library is free, multiplatform, has a Object-oriented C++ API, contains multiple numerical integrators, it supports serial

[1] Application Programming Interface

chains, closed chains and tree structured mechanisms. The currently implemented joints are the Ball, Rigid Z-transform (used for branching), revolute and prismatic joints.

The library is not mature enough, it is still in beta phase, is missing important joints such as hinges and it doesn't contain a collision detection.

Havok http://www.havok.com/

This commercial engine is designed for games, and is very successful, games such as Half-Life 2 are using it. The library is multiplatform, including consoles, it has an optimized collision detection and resolution (based on a variable step size), rigid body[2] dynamics, contains multiple ODE [3] solvers including Euler and RK45, multiple friction models, automatic deactivation of objects. All important joint types are supported: ball, hinge, car wheel, springs, rag doll, including joint limits. Continuous Physics was recently added, allowing handling of high-velocity game objects.

Meqon http://www.meqon.com/

This commercial Software Development Kit (SDK) seems very complete. It is multiplatform including consoles, has an Object-oriented C++ API, a built-in XML parser with full support for geometries, joints, etc. It contains an optimized collision detection for all geometry types, vehicle physics, automatic deactivation of objects. Important joint types are supported: ball, hinge, slider, prismatic, fixed, including breakable joints and joint limits.

Open Dynamics Engine http://ode.org

ODE performs as good as top-class commercial engines. It is a free software, multiplatform, features a C/C++ API, automatic deactivation of objects. It has several friction models, contains a collision detection for primitives (box, sphere, capsule) and triangle mesh. Ball, hinge, hinge-2 (car wheel), slider, univeral and contact joints are supported, as well as joint limits. It includes two methods for solving the constraints: a normal matrix inversion and an iterative method.

It is missing the cylinder geometry which is required, however this feature can be added manually since it is an open-source software.

[2]A rigid body is a non deformable solid body

[3]Ordinary Differential Equation

Novodex from AGEIA, http://www.novodex.com/

This is one of the most promising physics engines, namely because of their hardware solution (see http://www.ageia.com/technology.html), it is a commercial software, multiplatform, including consoles, features a high-speed collision detection and vehicle physics.

It is the only library supporting *Multi-threading*, which means it can take advantage of multiprocessors and next generation multicore CPUs. It also supports AGEIA *PhysX PPU*, the world's first Physics Processing Unit (PPU), an entirely new category of processor that promises to revolutionize gaming in the same way that the graphics processing unit (GPU) did in the 1990s.

Tokamak http://www.tokamakphysics.com/

This is a good free engine, but not at a commercial level as it lacks speed and features, for example it only has an iterative constraint solver, one friction model, and only two joints although they are the most important ones: ball and hinge. It includes collision detection for primitives (box, sphere, capsule) and triangle mesh, and supports breakable joints.

Newton Game Dynamics http://www.physicsengine.com/index.html

This library seems complete: it is free, multiplatform, contains three different solver mode configurations, two different friction models, a fast and accurate collision detection, and several friction models.

Vortex by CM-Labs, http://www.cm-labs.com/

Vortex is a proven working physics library targetted at scientists, it is a commercial solution, has a object-oriented C++ API. It contains several friction models, an accurate collision detection, vehicle dynamics. Many joints are supported including ball, hinge, car wheel, slider, prismatic, universal, fixed and spring joints, as well as breakable joints and joint limits. Additionally Vortex has an intergrated XML parser, allowing to load geometries, constraints, and more.

Karma http://www.renderware.com/physics.asp

Karma is a gaming version of Vortex tailored for computer games, optimized for speed but has less features, for example Karma does not have mesh collisions and does not allow rotational (inertial) effects.

Bytegeist Vehicle & Physics SDK http://www.bytegeistsoftware.com/

This is another commercial library, features a generic C API, multiplatform support, fast and accurate collision detections, a friction model, automatic deactivation of objects.

OpenTissue Project http://www.opentissue.org

This open-source framework contains a wide variety of algorithms and data structures, it was originally used as a playground for experiments. Today OpenTissue works as a foundation for research and student projects in physics-based animation at the Department of Computer Science, University of Copenhagen (commonly known as DIKU).

The physics engine we chose is ODE because it is at a mature stage, it is free, has a performance comparable to the best commercial packages, and has many similarities with Vortex, its creator having worked on it. That last point is important because many people have been using Vortex and switching to ODE instead of another library will be a smoother transition.

2.2 ODE analysis

ODE supports rigid body dynamics with an arbitrary mass distribution.

Bodies are connected to each other via joints. An "island" of bodies is a group that can not be pulled apart, every body is connected somehow to every other body in the island. Each island in the world is treated separately when the simulation step is taken. This is a crucial point for optimizing simulations.

A very stable first order integrator is used, meaning that the calculation errors does not cause the system to have non-physical behaviors. However it is not particularly accurate, not enough for quantitative engineering yet, unless the simulation step size is very small.

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

The available collision primitives are: sphere, box, capped cylinder, plane, ray, triangular mesh. We notice the absence of a normal cylinder and a heightmap, those will have to be added manually.

To speed up collision detection, it is important to reduce the number of geometries as much as possible, that is why a an obstacle is usually modelled as a single cylinder, and the wall as one box. This is especially true for the robots.

The friction model used is an approximation of Coulomb friction model.

Worldstep vs. Quickstep Two time stepping methods can be used. First, worldstep which solves the system of constraints by inverting a matrix, that takes a lot of time and memory, however the results are very accurate. The second method is called quickstep, it is an iterative constraint solver method, the number of iterations can be chosen and the accuracy/speed ratio can be controlled that way. Quickstep is only really efficient if all the connected bodies have the same mass like close to 1.0., that is a consequence of the iteratioms.

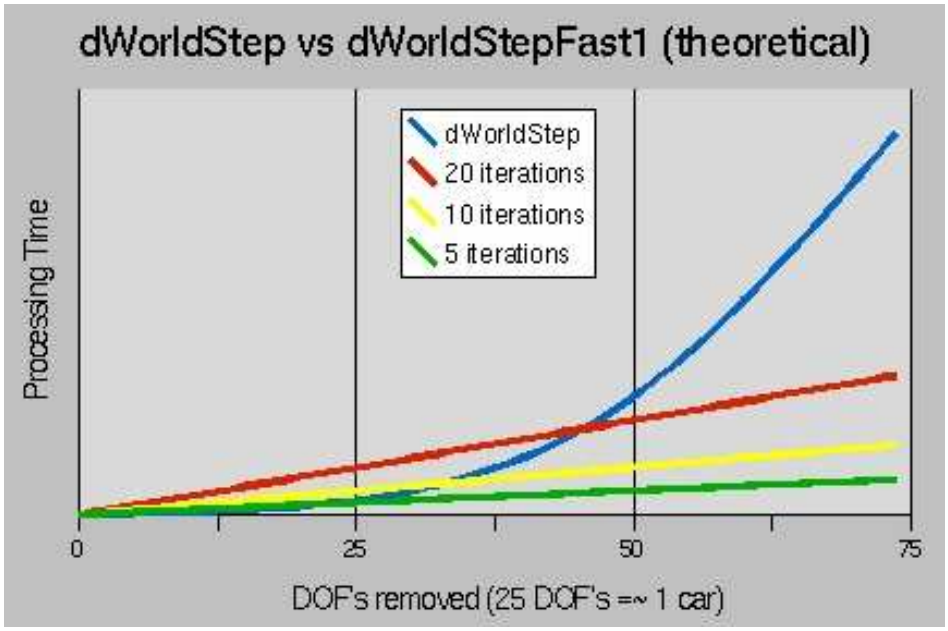


Figure 2.1: Speed advantage of Quickstep [ode doc]

Introducing a joint removes a certain number of degrees of freedom (DOF), for example the ball-and-socket joint removes three, and the Hinge five.

WorldStep running time is proportional to the cube of the number of DOFs removed during a step. Quickstep algorithm however, is roughly linear (see Figure 2.1). Memory requirements are propotional to the square of the number of DOFs for worldstep and still linear for quickstep (see Figure 2.2).

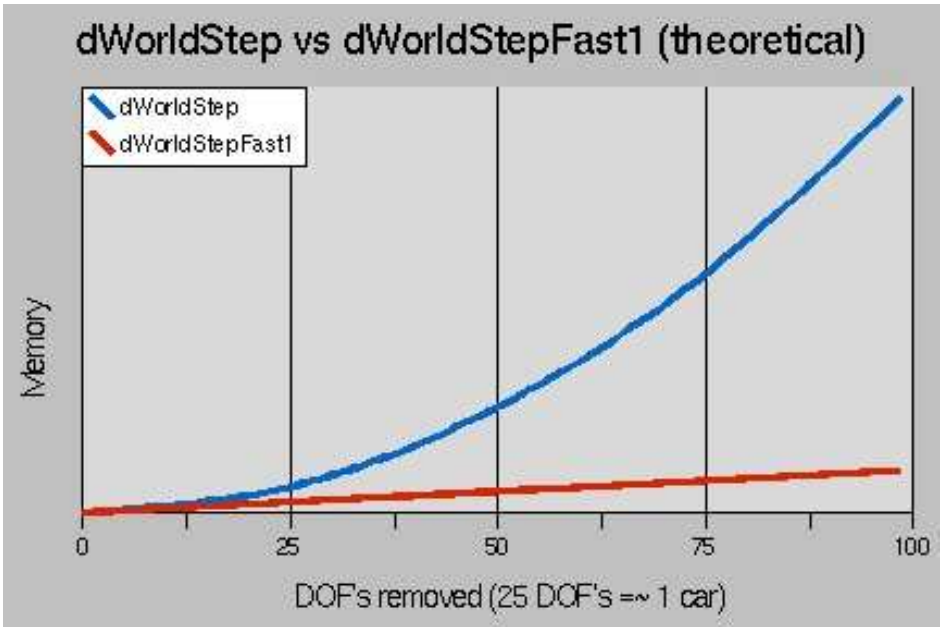


Figure 2.2: Memory advantage of Quickstep [ode doc]

We added a new method which tries to combine the best of both worlds by using quickstep only on the islands having too many joints. This feature has not been completely approved yet but the early tests are promising.

A good way to speed up the simulation is to model objects with as little bodies and joints as possible. One way to decrease the number of bodies is to use composite objects, ie multiple geometries are associated to only one body. Using simpler joints like a Ball-and-socket instead of a Hinge will reduce the number of DOFs removed and in turn increase the simulation speed. This is why several models of the robot have been created, one being simplified as much as possible while keeping all its abilities like the gripper and a rotating turret, the other one is a more detailed model.

2.3 Robot modeling

The simulation of a robot is not only limited to its movements, it must also be able to interact with the environment, via the sensors and the actuators.

2.3.1 Actuators

Actuators have been divided into two categories: joint actions and user actions.

Joint actions are used to control joints by setting a desired velocity. Every motor of the robot that corresponds to a simulated joint is controlled in this way.

User actions exist for actuators that can not be controlled automatically because they are not associated to a simulated joint. In this case, a C++ custom code is required. A default custom action is the enabling and disabling of objects: for example lights can be switched on and off in this way.

Another important concept is the emitter, that is a source that a sensor type can detect, for example a sound emitter can be detected by a sound sensor, but not by a light sensor. User actions can be associated to emitters, which allows a controller to modify properties of that emitter such as the sound intensity and frequency in the case of a sound sensor.

2.3.2 Sensors

Real robots are able to perceive the environment via their sensors, thus sensors have to be simulated as well. This can be a very difficult task in many cases so different techniques have been implemented, allowing to choose between speed and accuracy.

Sensing techniques are grouped in three categories:

- Sampling

  This technique relies on a look-up table containing readings taken from the real robot for each sensor. Two indexes are required to retrieve the sensor values in the look-up table: the distance between the robot and the sensed object, and the relative angle between the front direction of the robot and the object. A third index is used to retrieve a single sensor reading.

Because look-up tables are only indexed by a distance and one angle, they are two dimensional and so the sampling technique is limited to a flat plane.

The main advantage of using this technique is that no calibration is required to obtain similar values as the real robot since the samples were taken from it. The disadvantages is that those samples are needed for each different object to be perceived, and it is not always possible, for example taking samples from ground holes, and this technique only works on a plane.

It is easy to simulate the sensing of one object using that technique, the difficulty appears when one must sense many objects at the same time. The problem is how to combine the sensor values together. This will be explained in the next chapter.

- Simulated

  This technique involves calculations and a calibration to estimate real robot readings as accurately and as fast as possible. Proprioception such as the traction or joint sensing are exclusively simulated in this way, using direct calls to the underlying ODE engine.

- Raytracing

  It has two utilizations:

  - Visibility testing of an object, for example, a light. A ray is shot from the robot towards the object, if a collision occurs in between then the object is considered hidden, otherwise it is visible and in that case Sampling or Simulated techniques are used to retrieve the sensor value.
  - Get the magnitude, for example in proximity sensing. A ray is shot in the direction of the sensor, if an intersection is found then the sensor value will be greater than zero, it will depend on the distance between the source of the ray and the collision point.

  Raytracing has the advantage of working everywhere including on a 3D terrain, it can also sense any object geometry. The disadvantage is that it is slower than other techniques especially when using a high number of rays.

On every sensor, a uniform noise can be applied to simulate the real sensor noise.

Chapter 3

S-bot modeling

In this chapter, we describe how the general concepts of actuators and sensors mentioned previously have been applied to the simulation of an s-bot.

3.1 Actuators

The elevation motor, the rotation motor, the gripper motor and the differential treels drive are all simulated with a joint action (see section 2.3.1). That is, no programming is required at all.

The eight RGB LEDs are simulated with a user action and a light emitter. The two loud-speakers are simulated with a user action and a sound emitter.

3.2 Sensors

For this work, we are interested in simulating the proximity sensors, the ground sensors, the light sensors, the microphones, the traction sensor, the light barrier within the gripper, and the 3-axis inclinometer. The camera sensor and the humidity/temperature sensors are left for future work.

In the following sections we will give a brief description of the sensing devices and we will present how they are simulated.

3.2.1 Proximity sensors

- Description

The s-bot has 15 infra-red (IR) proximity sensors that can detect obstacles in its immediate neightborhood. Sensors are not uniformly placed on the turret, but approximately with 24 degrees between them, however there is no sensor at the location of the gripper.

IR 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 volume that can be approximated by a cone with an angle of 40 degrees and a range of 20cm.

- Modeling

  1. Sampling

     Sensing of one object with that technique has been described in Section 2.3.2. The only difficulty is now to combine the values when sensing multiple objects at the same time.

     The solution found is to sense every object around the robot without performing any visibility test, and then keep the maximum values of each sensor. This way is quite fast and accurate.

  2. Raytracing

     This is obviously an approximation of the reality because the real s-bot has a cone of vision of 40 degrees while shooting one ray means a zero degrees vision. To counter this, we can have one ray moving randomly inside the cone of vision, or have more than one ray, for example a central ray and four peripherical rays.

  3. Simulated

     This is a very fast approximation of raytracing that shoots 360 rays around the robot, however it only works with a 2D plane. Also each object needs to be approximated by a bounding volume, either a bounding cylinder or an object-oriented bounding box (OOBB), this is used for a wall for example. See the appendix for an explanation of the shadows algorithm.

There are some issues: simulated IR sensors suffer from the "dead spot", which is the space in which small objects remain undetected. One way to solve this is using more rays, or moving rays. Another problem is given by the fact that the real sensors are sensitive to the color of the reflecting surface. This is not taken into account in simulation.

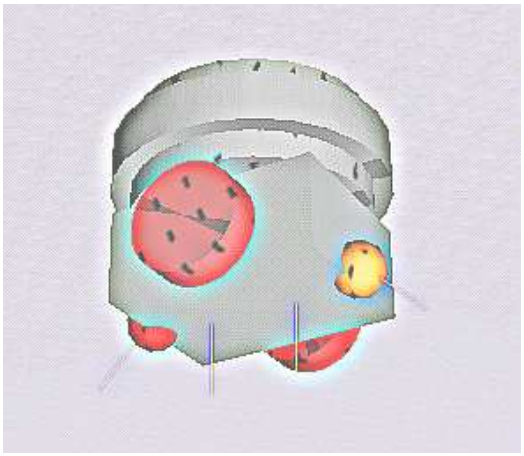


Figure 3.1: Ground sensors

3.2.2 Ground sensors

- Description

  The s-bot has 4 ground sensors placed under the treels body, those are useful to detect terrain conditions like holes and edges.

- Modeling

  In this case, sampling is hardly feasible, holes and edges can have too many shapes, so the raytracing is used exclusively.

  Ground sensors are modeled as ray sensors, similarly to the proximity sensors. They can not distinguish color either and also suffer from the "dead spot" issue, however it is not so crucial due to the very limited sensing range.

3.2.3 Light sensors

- Description

  The s-bot features 8 light sensors able to sense the intensity of a light source.

- Modeling

  Raytracing here is used for a visibility test: a ray sensor is placed between the robot and every light source. At every time step intersections

are computed. If there is no intersection on the ray sensor then the light is visible. This is a very fast approximation of the reality because only one ray is used and the light is considered as a point.

From there, either the distance is converted to a sensor value, with the intensity of the source being weighted by the inversed square distance, or the sampling technique is used, which has the advantage again to not require any rescaling of the sensed values.

In case of multiple light sources, sensor values need to be combined, this is done by summing their values together, and then applying a sigmoidal saturation function (see Figure 3.2):

$y = \frac{2}{1+\exp(-x)} - 1$

3.2.4 Sound sensors

- Description

  A real s-bot hosts four microphones on top of its turret. The sound device is intended for simple communication and localization. S-bots can sense sound intensity on a certain frequency range.

- Modeling

  Sound sensing is difficult to simulate because of reflection of sound waves on the objects present in the environment. A simplified model is used in which the s-bot only has one sound sensor and where every sound reaches the sensor, ignoring reflections.

  1. Sampling

     A look-up table similar to the proximity and light, is used to find sensor values depending on the relative orientation and position of the robot with the sound source.

  2. Simulated

     The sensed values are approximated by weighting the sound intensity by the inverse squared distance between the microphone and the source.

  In case of multiple sound sources, sensor values need to be combined, this is done by summing their values together, and then applying a sigmoidal saturation function:

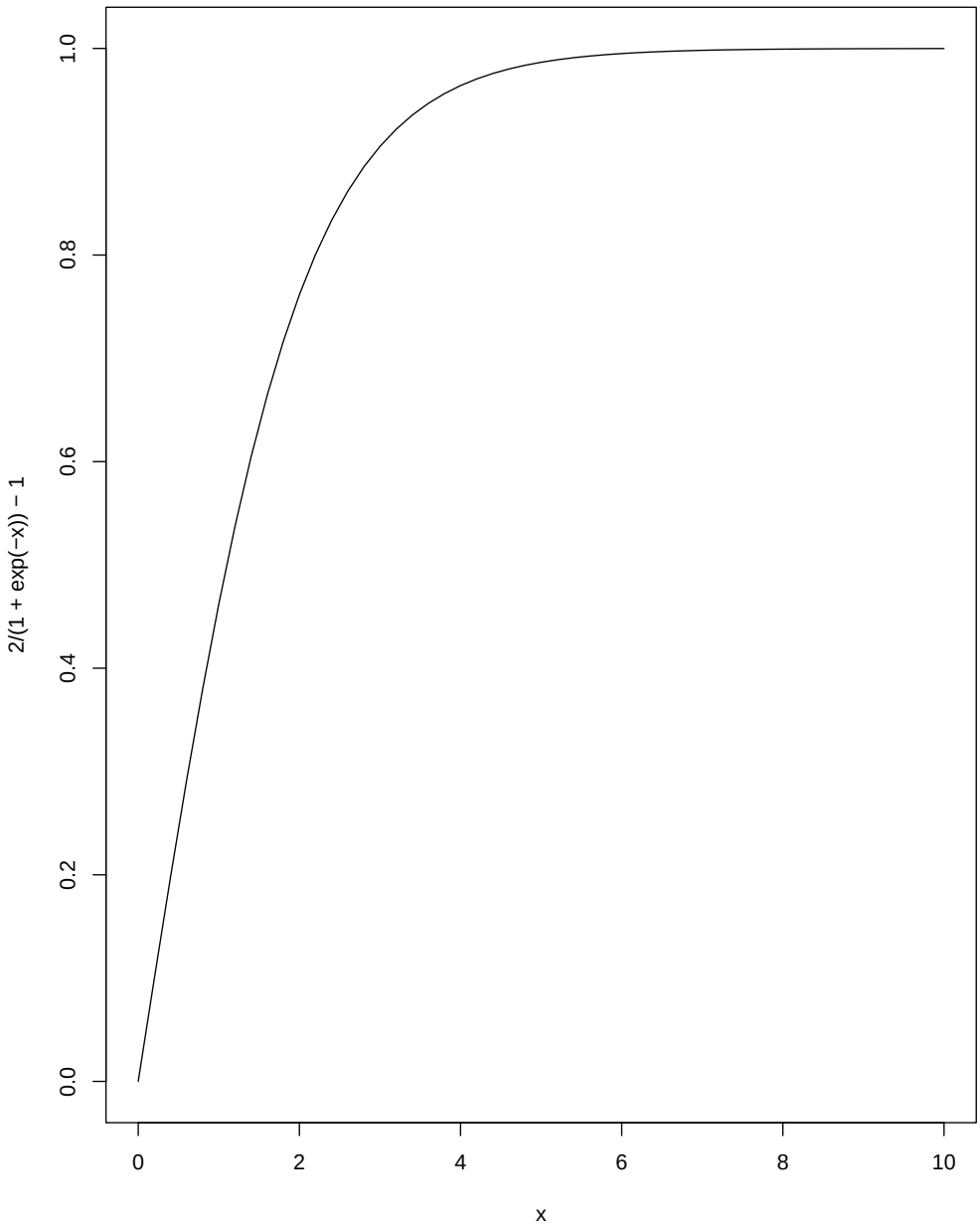


Figure 3.2: Sigmoidal saturation function

$y = \frac{2}{1+\exp(-x)} - 1$

Light barrier within the gripper

- Description

  A light barrier is placed within the gripper, it consists on one light emitter and one sensor. When an object interrupts the light, the sensor does not detect it anymore, which means that a secure grasp can be executed.

- Modeling

  There are two methods available to simulate the light barrier:

  1. Model sensor: collisions are detected with a dummy [1], placed inside the gripper, the object colliding it is considered grippable. If the dummy geometry is a ray then we have a close representation of the reality.
  2. Simple: this only looks at distances and orientations between a virtual gripper and other objects, those objects require to be grippable, this is the case when they have a model emitter.

3.2.5 Joint sensors

- Description

  A joint sensor can sense forces at the two sides of a joint, this is used to simulate the traction sensor.

- Modeling

  ODE computes the forces and torques at the joints automatically. Therefore it is sufficient to read the values computed by ODE.

[1] We define the dummy as an object that does not interact with anything but can remember what objects it virtually collided with, it could be called a phantom object instead.

Chapter 4

Software design

In this chapter we present the main classes (Section 4.1) including the UML class diagram. We will have an overview of the main loop (Section 4.2). Finally, we will discuss the use of XML in the simulator (Section 4.3).

4.1 UML

IEnvironment This is the main application class, it handles the world initialization and destruction, the main loop, the command line, the control loop, the drawing, the keyboard controls, the camera movement and orientation via the mouse.

It contains a list of objects, materials, contacts and samples. Lists have been chosen instead of arrays to handle adding/removing objects in real time and without limit.

Every object added to the list will be activated and drawn, the material list contains all the loaded materials, the contact list contains all the possible contacts between every material, this is used during the collisions, the sample list contains all the loaded samples used for the sampling sensing technique, only the ones used in the simulation are loaded, so there is no waste of memory.

Model This is the smallest object, it contains a body and a geometry required for the dynamics simulation and the collisions. It also contains a material and a list of renders, used to draw it. The model handles the collisions, they only occur between models.

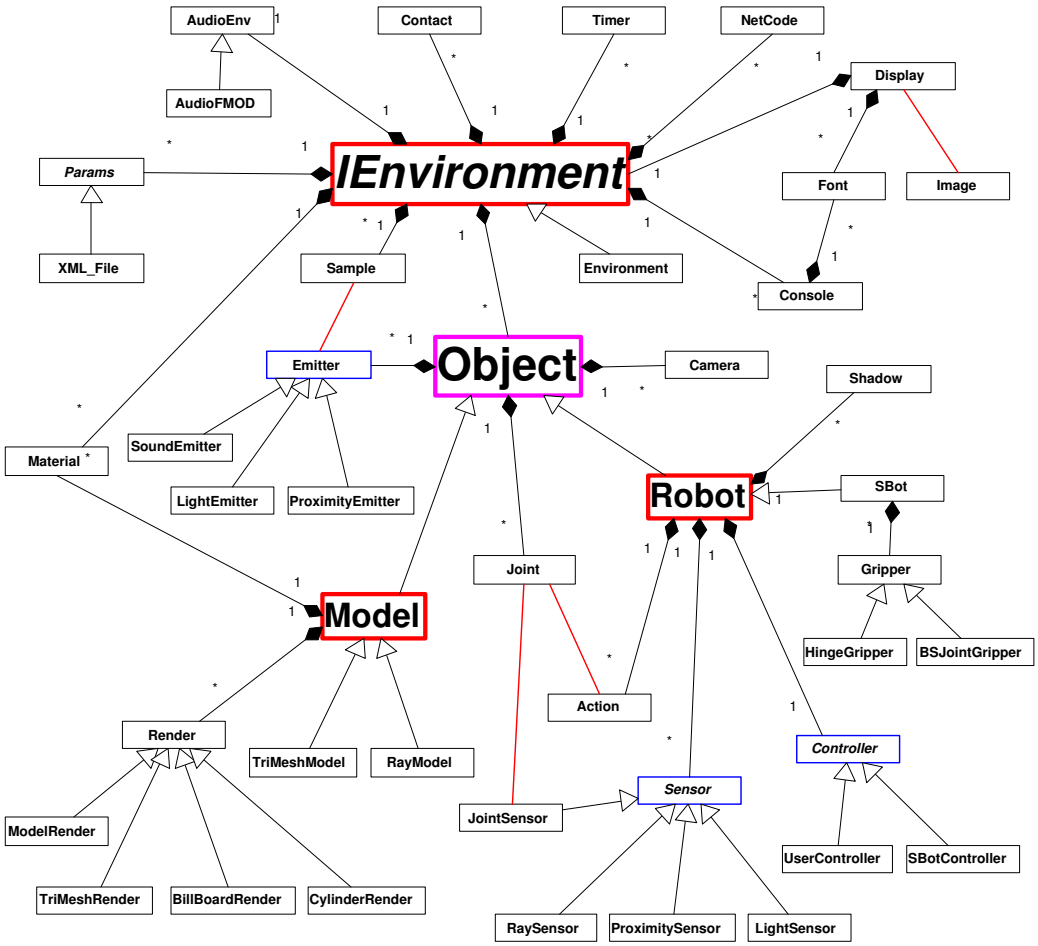


Figure 4.1: Simplified UML class diagram

Object The object is a general hierarchical object, as it can contain other objects, the only object that can not contain anything is the model.

Objects contain a list of cameras, emitters, history, parts and joints. Cameras are used to follow the object from different viewpoints; emitters are required to make the object detectable; history stores all the previous positions of the object; parts are the children objects; joints are the list of joints between the parts.

Robot A robot is an object that also contains a controller, and a list of sensors and actions.

Its main function is 'ApplyOutputs' which takes the inputs from the sensors, converts them into outputs using the controller, then applies those outputs to the actions.

Sbot The sbot is a robot that also contains a gripper. It is a fully customizable class, the main function is 'CustomAction' which receives an action number as defined in the XML file.

Controller This converts the inputs from the sensors into outputs to be sent to the actuators. The controller is user-defined and it can consist in a behavior based module, a neural network or any other suitable algorithm.

Sensor Used as inputs for the robots. See Section 3.2 for details about the implemented sensors.

Emitter The emitter allows an object to be sensed by a sensor. Emitters can load samples from a file, those will then be used for the sampling sensing technique.

Action Converts outputs from the robot into an action like a movement, or the closing of a gripper.

There are two types of action: joint and user. The joint action allows to move any joint in any possible way, or modify joint parameters. The user action does not do anything but rescaling the output, and it is processed by the robot or sbot CustomAction method.

Render This is used to render a model, many classes are defined allowing to render either the model geometry or another geometry. Example a ball model could be rendered with a cylinder render and then will appear like a cylinder on the screen although it is a sphere. This is used for lights for example, they are rendered as a billboard (an image facing the camera). Everything can be defined in the XML and also models can have several renders if required.

Joint This is a breakable joint, it records the forces at each side of the joint, these values are requested by the joint sensor.

XML_File This class imports and exports XML files, creates the objects and adds them to the environment (see Section 4.3).

Display This is the 3D rendering class based on OpenGL, it can draw object primitives, initialize extensions, resize the window to fullscreen, save a screenshot. It contains a texture manager, useful to not have a same texture loaded several times.

Console A classic console that supports binds, aliases and commands, it is a part of the interface and is useful for debugging.

4.2 Main loop

Here is the mainloop:

1. Handle SDL events (keyboard, mouse, network)
2. Keyboard Controls
3. Check if joints are broken in any object
4. Robot Controls
    (a) Sense: get inputs from all sensors
    (b) Control
        i. Controller converts inputs to outputs
        ii. Apply outputs using the actions

5. Simulation
   (a) Preprocessing
   (b) ODE Collisions: use adequate contacts for each collision
   (c) ODE Simulation: body dynamics + constraints system solved
   (d) Postprocessing: update AABB
6. Draw everything using the object list
7. Update audio and network

4.3 XML

XML plays an important role in the simulator: the description of the 3D scene and the objects is done in the XML file, that includes bodies, geometries, models, constraints, renders as well as the materials and contacts used. This is a standard way adopted by many simulators or physics engines.

However here, the XML also describes the controllers, actuators, sensors and emitters with a lot of possible parameters for each. This allows users to create and control their own robots via the keyboard without a single line of C++ code. Of course code will be required as soon as a new controller is desired.

See Appendix A for more details.

Chapter 5

Testing

In this chapter we show the results obtained with several benchmark tests (Section 5.1). In section 5.2 we describe four examples of simple controllers for the s-bots.

5.1 Benchmarks

Benchmarks are needed to test the speed of different components of the simulator, to see which integrating methods are faster and under which circumstances (Section 5.1.1 and 5.1.2), and also to evaluate the sensing method speed (Section 5.1.3 and 5.1.4).

All the tests were ran using a standard computer provided with an AMD Athlon 2800+ XP processor with 512 MB of memory, running a Linux Debian distribution.

Each benchmark has been ran more than 100 times with a different seed. Rendering was disabled and no other application was running in the background, the tests lasted 1000 simulation frames, with a step size of 0.1 second which means that 100 seconds were simulated. The average execution time of the simulation is taken as evaluation meter. The presented values are in seconds.

5.1.1 Random movement

Eight different tests were ran, with an increasing number of robots each time, they had no sensors and used a special controller to move randomly. The environment only consists on a plane, robots can potentially collide with the

Number of s-bots	worldstep	quickstep	world ratio	quick ratio
1	0.08638	0.41086	1157	243
2	0.16606	0.81016	602	123
4	0.32694	1.62109	305	61.7
8	0.65303	3.25002	153	30.8
16	1.40601	6.50322	71.1	15.4
32	4.16268	13.1822	24.0	7.59
64	15.0444	27.4801	6.65	3.64
128	52.1648	57.5374	1.92	1.74

Table 5.1: Random movement

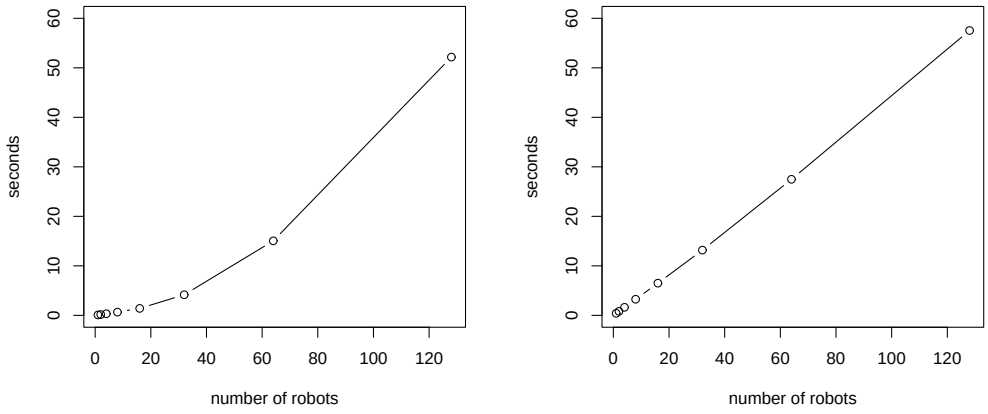


Figure 5.1: Random movement. Worldstep (left) Quickstep (right)

plane and with each other only. Two ode-specific integration methods are used: worldstep and quickstep (see Section 2.2).

Since 100 real seconds are simulated, it means that simulating 128 robots goes about twice faster than real time speed, simulating only 8 would be 150 times faster. Remember that worldstep complexity increases with the cube of number of joints per island (group of joined objects), while quickstep increases linearly. When a collision occurs in ODE, a contact joint is created between the two collided geometries to ensure a separation between them. When the number of robots is low, few collisions are being made, hence few contact joints, this explains why worldstep performs significantly better than quickstep. However quickstep catches up as the number of collisions increases

Number of s-bots	worldstep	quickstep	world ratio	quick ratio
1	0.08638	0.41086	1157	243
2	0.41023	0.84053	244	119
4	2.72198	1.73786	36.7	57.5
6	8.90234	2.61219	11.2	38.3
8	24.8694	3.49186	4.02	28.6
12	75.4717	5.26593	1.33	19.0
16		7.06165		14.2
20		8.88494		11.3
25		11.0939		9.01

Table 5.2: Connected robots

as seen when simulating 128 robots.

If we were using a good controller allowing robots to not collide with each other, the worldstep would really outperform quickstep all the time.

We conclude that worldstep should be used in scenarios not involving too many collisions or contacts.

5.1.2 Connected robots

This benchmark consists in s-bots connected to each other in a defined formation. Nine tests were ran with an increasing number of robots, always grouped together, with worldstep and quickstep methods. In this test, robots use a slightly different controller, they move randomly but they also grip each other if they can. Their initial position and orientation allows them all to grip another robot. The environment is still a plane.

The number of joints per island is higher here as there is just one island, which is the group of robots connected together, so the worldstep method literally explodes starting from 8 connected robots, memory usage increases quadratically with the number of robots and it even crashes with 16 connected robots because of this.

Quickstep method, is slower with one and two robots (see the previous benchmark) but catches up very quickly and offers a linear complexity performance. Having 100 physical seconds here, simulating a group of 25 robots connected together runs at 9 times the real time speed.

We conclude that quickstep should be used to simulate connected robots.

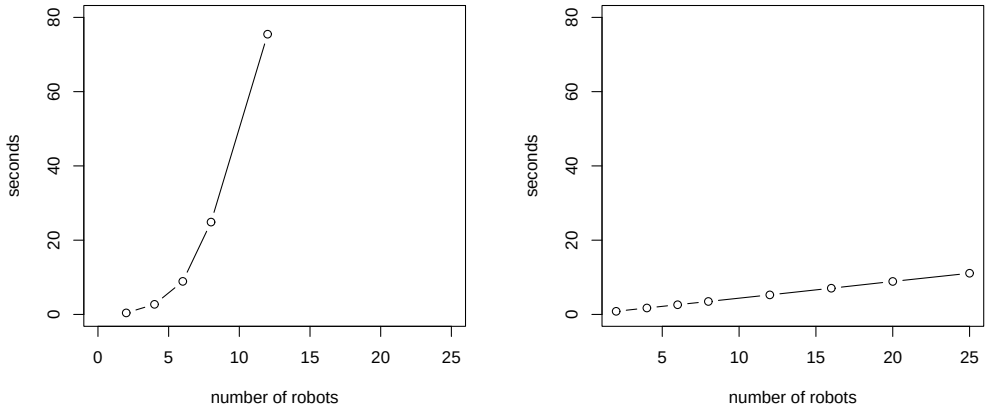


Figure 5.2: Connected robots. Worldstep (left) Quickstep (right)

Number of s-bots	raytracing	sampling	shadow
1	0.17729	0.17282	0.15556
2	0.22601	0.19191	0.17658
4	0.33981	0.22606	0.21904
8	0.57722	0.30498	0.30375
16	1.34214	0.47401	0.54493
32	4.04531	1.09081	1.55296
64	12.8800	3.16847	3.63465

Table 5.3: Proximity sensors

5.1.3 Proximity performance test

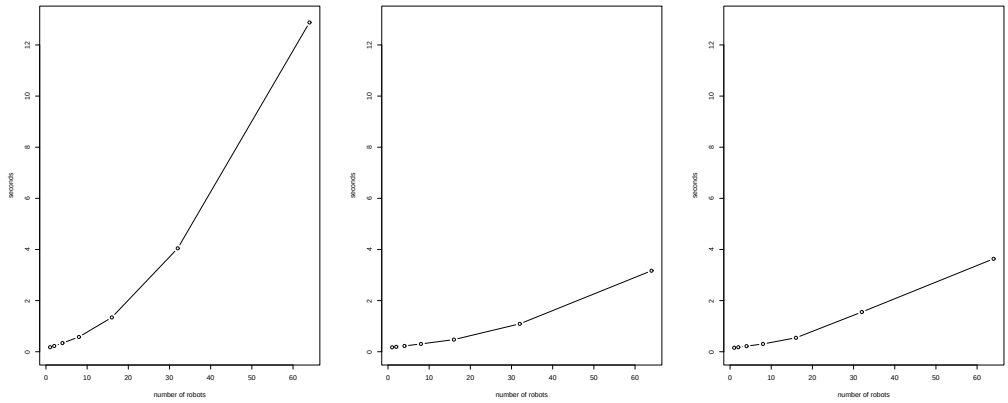
The three proximity sensing methods are benchmarked, we expect raytracing to be the slowest method because it must collide rays against the environment, while sampling and shadow methods require an iteration over all the objects, plus basic calculations for sampling, moderate for shadow.

The number of ray sensors used for the raytracing is 15.

Robots do not move as we want to test only the sensors and not the simulation, robot bodies themselves become inactive and do not use any computing power.

The environment contains a plane, 4 walls and 24 obstacles.

Results confirm the expectations but it is interesting to see that raytracing



Number of s-bots	light	sound
1	0.24113	0.14983
2	0.35754	0.16325
4	0.55386	0.18549
8	1.11743	0.23658
16	2.85470	0.36204
32	7.78998	0.73499
64	27.5786	2.16282

Table 5.4: Sound and light sensors

is actually going quite fast and is not slowing down the simulation so much with 15 rays per robot, so a total of 960 rays were used in the last test.

5.1.4 Sound and Light performance test

This last test analyzes sound and light sensing performance. Robots do not move because we want to test only the sensors and not the simulation, robot bodies themselves become inactive and do not use any computing power.

The environment contains a plane, 4 walls and 24 light/sound sources.

The light sensor uses a ray for visibility testing while the sound sensors does not need one, this explains the difference of speed between the two.

5.2 Experiments

Four experiments were run with a random number of s-bots, to test the functioning of various sensors and actuators. The first three experiments take place on a flat ground with four walls and several obstacles, the last one on a 3D rough terrain. The controller used is based on Braitenberg studies.

In the book "Vehicles: Experiments in Synthetic Psychology", Valentino Braitenberg describes a series of thought experiments in which vehicles with simple internal structure behave in unexpectedly complex ways. He describes simple control mechanisms that generate behaviors that, if we did not already know the principles behind the vehicles' operation, we might call aggression, love, foresight and even optimism[3].

Braitenberg vehicles are made up of two wheels (left and right), a number of sensors mounted on the front of the vehicle, and connections from the sensors to the motors, these connections are analogic and either increase or decrease power to the motors.

5.2.1 Braitenberg obstacle avoidance

With its sensors, the s-bot can detect an obstacle on its front-left and on its front-right, the closer the obstacle the higher the value of the sensor. If an obstacle is detected on its left, the s-bot will slow its right wheel down, thus will perform a basic obstacle avoidance (see Figure 5.3).

Here is the pseudo-code for the controller:

```
ObstacleAvoidanceController(input,output)
{
  CalculateSides(input,side)
  output[left ]=1-side[right]
  output[right]=1-side[left ]
}
```

5.2.2 Braitenberg and phototaxis

This is the same as the previous experiment, but s-bots are now attracted by the light, this time if a light is detected on the left, the s-bot will slow its left wheel down in order to get closer. They deal with obstacles the same way as before.

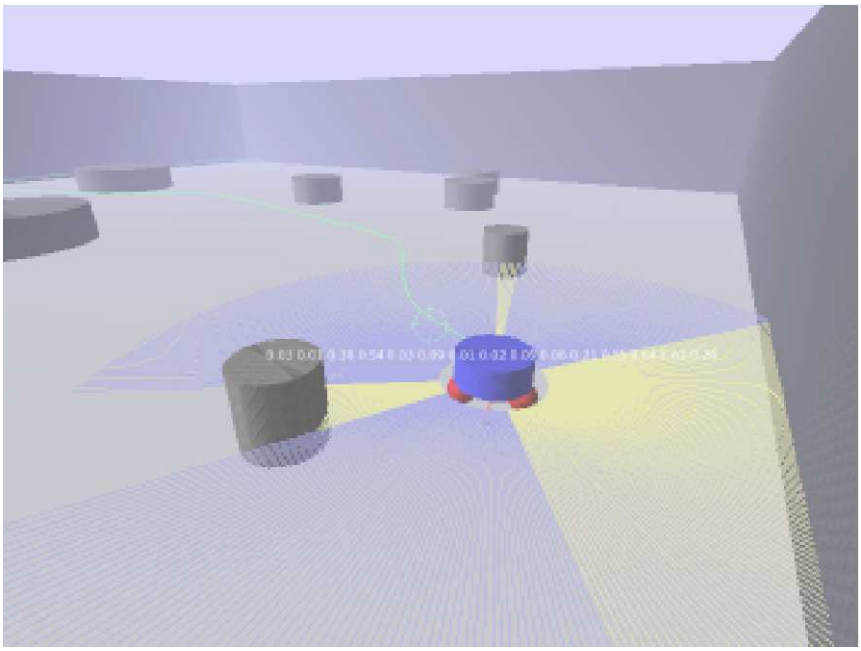


Figure 5.3: obstacle avoidance

Here is the pseudo-code for the controller:

```
LightObstacleController(input,output)
{
  CalculateSides(input,side)
  CalculateLights(input,light)
  output[left ]=1-side[right]-light[left ]
  output[right]=1-side[left ]-light[right]
}
```

We see that robots are attacted by the light, when both lights are on, two groups of robots are formed and immobile (see Figure 5.4), then we switch a light off, and we see that the robots are now gathered around the only light that is still on (see Figure 5.5), when we switch off the last light, robots go away looking for another light source (see Figure 5.6).

5.2.3 Braitenberg, phototaxis and assembly

This is the same experiment as the previous one but the s-bot also grips any object it can.

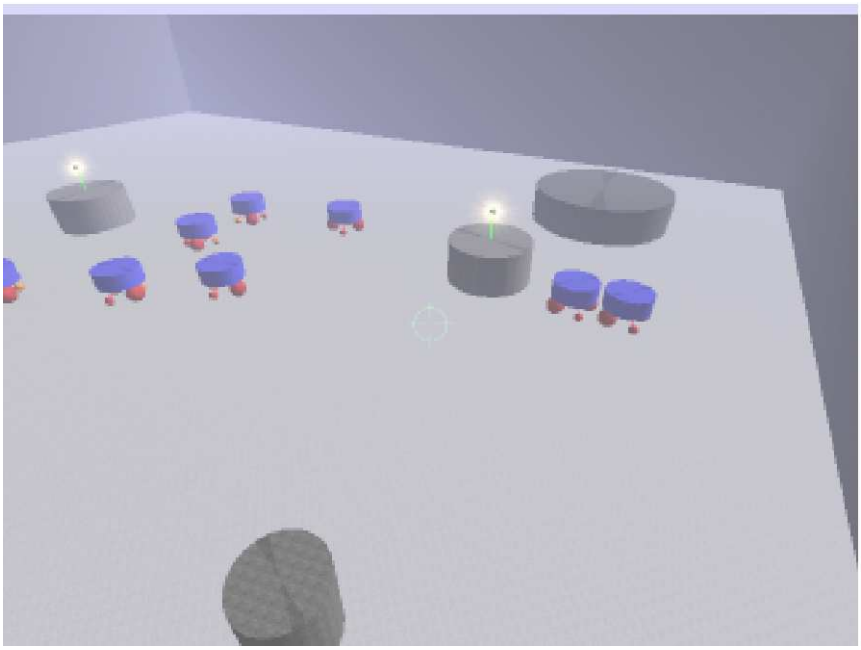


Figure 5.4: phototaxis, both lights are on, there are two groups of robots

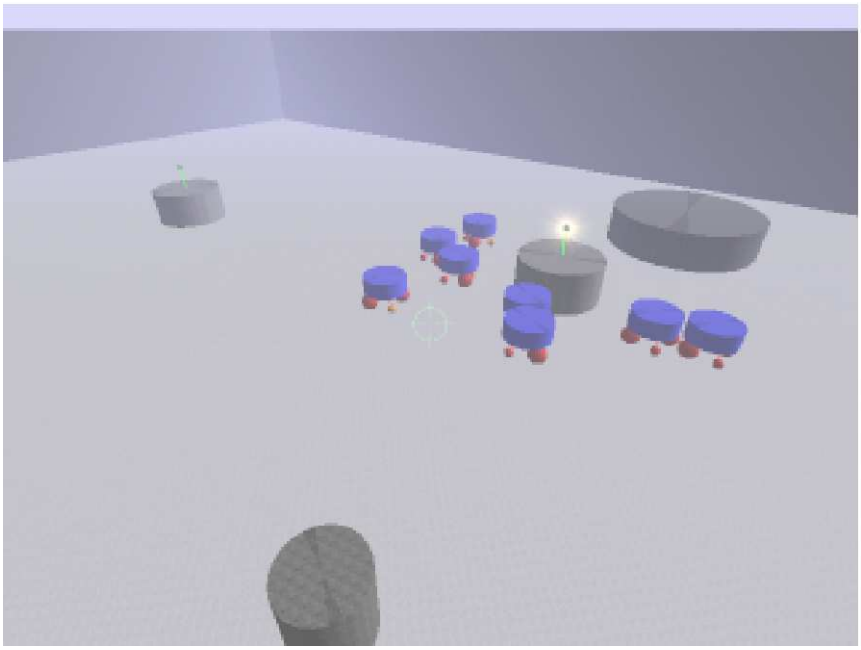


Figure 5.5: phototaxis, one light is on, there is one group of robots

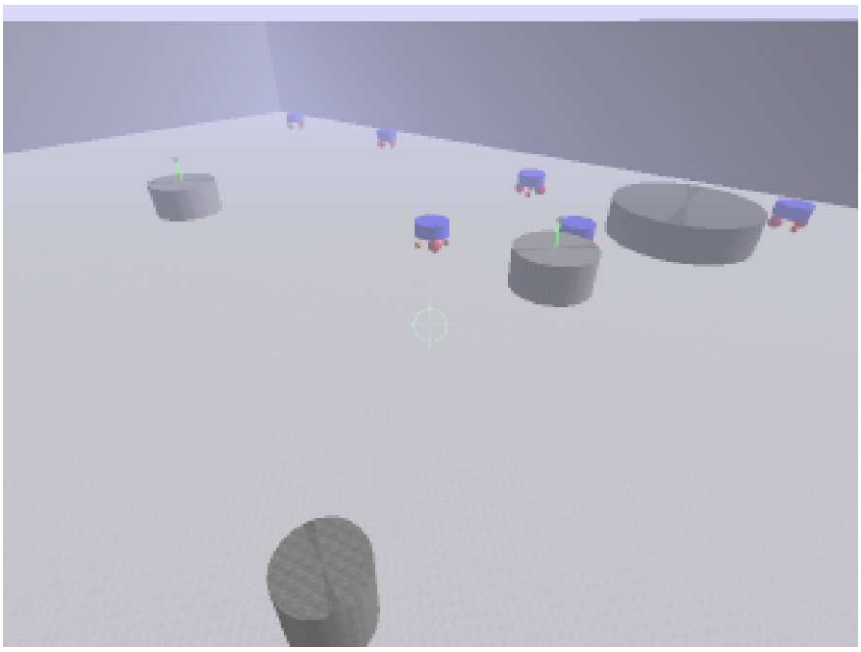


Figure 5.6: phototaxis, both lights are off, robots are spread

Here is the pseudo-code for the controller:

```
GripLightObstacleController(input,output)
{
  CalculateSides(input,side)
  CalculateLights(input,light)
  CalculateGrippable(input,grippable)
  output[left ]=1-side[right]-light[left ]
  output[right]=1-side[left ]-light[right]
  output[grip ]=grippable
}
```

5.2.4 Rough terrain

This experiment tests the obstacle avoidance controller on a 3D terrain, thus only raytracing proximity sensors can be used. The robot has difficulty to move in certain areas (see Figure 5.7).

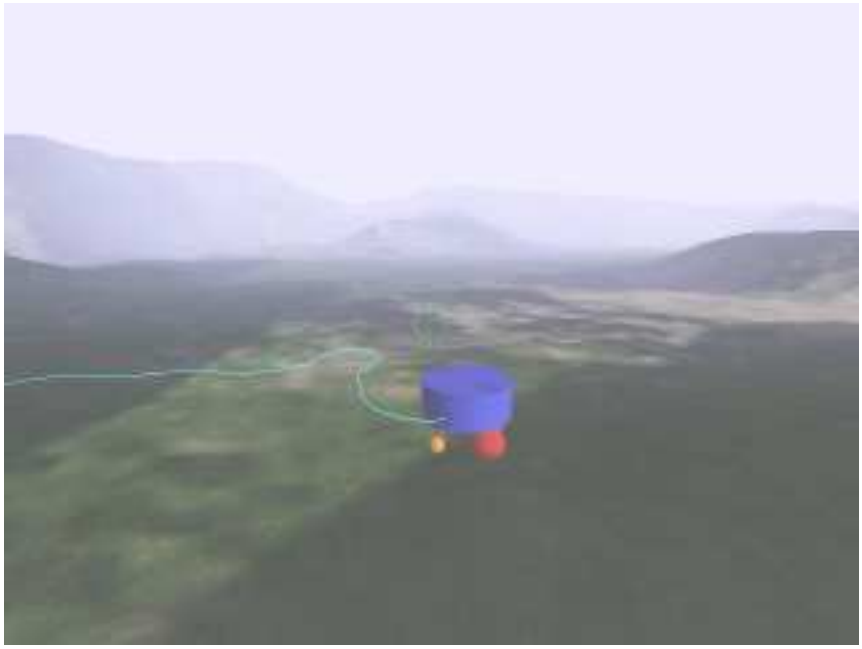


Figure 5.7: Rough terrain

Chapter 6

Conclusions

6.1 Achievements

We have presented in this work a simulator capable of handling a group of robots alone or connected together, with relatively good speed and accuracy.

After an analysis of the real s-bot, all its required sensors simulations have been implemented and validated through a set of benchmarks and experiments including obstacle avoidance, phototaxis, and movement on a rough terrain. An intuitive user interface has been developed, it features 3D rendering with special effects, a console for easy debugging as well as fully customizable keyboard and mouse controls. The simulations and program parameters are completely configurable via XML. Compilation has been run successfully on Windows XP and Debian Linux.

6.2 Future developments

Several sensors are missing and should be implemented in the future such as the camera sensor and the temperature sensor. Speed optimizations are required for the rough terrain.

Bibliography

[1] G.Baldassarre, S. Nolfi, and D. Parisi. Evolution of collective behavior in a team of physically linked robots. In R. Gunther, A. Guillot, and J.-A. Meyer, editors, *Proceedings of the Second European Workshop on Evolutionary Robotics*, pages 581-592. Springer Verlag, Berlin, Germany, 2003.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, NY, 1999.

[3] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, MA, USA, 1984.

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

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading. MA, 1989.

[6] F. Mondada, G. C. Pettinaro, A. Guignard, I. V. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, and M. Dorigo. SWARM-BOT: A new distributed robotic concept. *Autonomous Robots, Special Issue on Swarm Robotics*, 2004. To appear.

Chapter 7

Appendix A: Documentation