

Webots User Guide

release 5.0.10

copyright © 2005 Cyberbotics Ltd. All rights reserved.
www.cyberbotics.com

November 16, 2005

copyright © 2005 Cyberbotics Ltd. All rights reserved.
All rights reserved

Permission to use, copy and distribute this documentation for any purpose and without fee is hereby granted in perpetuity, provided that no modifications are performed on this documentation.

The copyright holder makes no warranty or condition, either expressed or implied, including but not limited to any implied warranties of merchantability and fitness for a particular purpose, regarding this manual and the associated software. This manual is provided on an *as-is* basis. Neither the copyright holder nor any applicable licensor will be liable for any incidental or consequential damages.

This software was initially developed at the Laboratoire de Micro-Informatique (LAMI) of the Swiss Federal Institute of Technology, Lausanne, Switzerland (EPFL). The EPFL makes no warranties of any kind on this software. In no event shall the EPFL be liable for incidental or consequential damages of any kind in connection with use and exploitation of this software.

Trademark information

Aibo™ is a registered trademark of SONY Corp.

GeForce™ is a registered trademark of nVidia, Corp.

Java™ is a registered trademark of Sun Microsystems, Inc.

Khepera™ and Koala™ are registered trademarks of K-Team S.A.

Linux™ is a registered trademark of Linus Torwalds.

Mac OS X™ is a registered trademark of Apple Inc.

Mindstorms™ and LEGO™ are registered trademarks of the LEGO group.

Pentium™ is a registered trademark of Intel Corp.

Red Hat™ is a registered trademark of Red Hat Software, Inc.

Visual C++™, Windows™, Windows 2000™ and Windows XP™ are registered trademarks of Microsoft Corp.

UNIX™ is a registered trademark licensed exclusively by X/Open Company, Ltd.

Foreword

Webots is a three-dimensional mobile robot simulator. It was originally developed as a research tool for investigating various control algorithms in mobile robotics.

This user guide will get you started using Webots. However, the reader is expected to have a minimal knowledge in mobile robotics, in C, C++ or Java programming and in VRML97 (Virtual Reality Modeling Language).

Webots 5 features a new layout of the user interface with many facilities integrated, like a source code editor.

If you have already worked with Webots 4, your existing world files and programs do not need to be modified for use with Webots 5.

We hope that you will enjoy working with Webots 5.

Thanks

Cyberbotics is grateful to all the people who contributed to the development of Webots, Webots sample applications, the Webots User Guide, the Webots Reference Manual, and the Webots web site, including Yvan Bourquin, Jordi Porta, Emanuele Ornella, Yuri Lopez de Meneses, Sébastien Hugues, Auke-Jan Ijspeert, Jonas Buchli, Alessandro Crespi, Ludovic Righetti, Julien Gagnet, Lukas Hohl, Pascal Cominoli, Stéphane Mojon, Jérôme Braure, Sergei Poskriakov, Anthony Truchet, Alcherio Martinoli, Chris Cianci, Nikolaus Correll, Jim Pugh, Yizhen Zhang, Anne-Elisabeth Tran Qui, Lucien Epinet, Jean-Christophe Zufferey, Aude Billiard, Ricardo Tellez, Gerald Foliot, Allen Johnson, Michael Kertesz, Simon Garnier and many others.

Moreover, many thanks are due to Prof. J.-D. Nicoud (LAMI-EPFL) and Dr. F. Mondada for their valuable support.

Finally, thanks to Skye Legon, who proof-readed this guide.

Contents

1	Installing Webots	15
1.1	Hardware requirements	15
1.2	Installation procedure	15
1.2.1	RedHat Linux i386	15
1.2.2	Windows XP	16
1.2.3	Mac OS X, version 10.3	16
1.3	Registration Procedure	17
1.3.1	Webots license	17
1.3.2	Registering	18
2	Getting Started with Webots	21
2.1	Introduction to Webots	21
2.1.1	What is Webots ?	21
2.1.2	What can I do with Webots ?	21
2.1.3	What do I need to use Webots ?	22
2.1.4	What is a world ?	23
2.1.5	What is a controller ?	23
2.2	Launching Webots	23
2.2.1	On Linux	23
2.2.2	On Mac OS X	24
2.2.3	On Windows	24
2.3	Main Window: Menus and buttons	24
2.3.1	File menu and shortcuts	24

2.3.2	Edit menu	26
2.3.3	Simulation menu and the simulation buttons	27
2.3.4	Wizard menu	28
2.3.5	Help menu	29
2.3.6	Navigation in the scene	29
2.3.7	Moving a solid object	29
2.3.8	Selecting a solid object	30
2.4	Scene Tree Window	30
2.4.1	Buttons of the Scene Tree Window	30
2.4.2	VRML97 nodes	32
2.4.3	Webots specific nodes	33
2.4.4	Principle of the collision detection	34
2.4.5	Writing a Webots file in a text editor	34
2.5	Citing Webots	35
2.5.1	Citing Cyberbotics' web site	35
2.5.2	Citing a reference journal paper about Webots	36
3	Tutorial: Modeling and simulating your robot	37
3.1	My first world: kiki.wbt	37
3.1.1	Setup	37
3.1.2	Environment	38
3.1.3	Robot	41
3.1.4	A simple controller	50
3.2	Adding a camera to the <i>kiki</i> robot	53
3.3	Adding physics to the <i>kiki</i> simulation	54
3.3.1	Overview	54
3.3.2	Preparing the floor for a physics simulation	55
3.3.3	Adding physics to the <i>kiki</i> robot	55
3.3.4	Adding a ball in the <i>kiki</i> world	56
3.4	Modelling an existing robot: pioneer2.wbt	56
3.4.1	Environment	56

<i>CONTENTS</i>	9
3.4.2 Robot with 16 sonars	58
3.4.3 Controller	65
3.5 Transfer to your own robot	65
3.5.1 Remote control	65
3.5.2 Cross-compilation	66
3.5.3 Interpreted language	67
3.6 Adding custom ODE physics	68
3.6.1 Introduction	68
3.6.2 Files	68
3.6.3 Implementation	68
3.6.4 Compiling the shared library	70
3.6.5 Example	71
4 Robot and Supervisor Controllers	73
4.1 Overview	73
4.2 Setting Up a Development Environment	73
4.2.1 Under Windows	73
4.2.2 Under Linux	75
4.2.3 Under Mac OS X	75
4.3 Setting Up a New Controller	76
4.4 Webots Execution Scheme	76
4.4.1 From the controller's point of view	76
4.4.2 From the point of view of Webots	77
4.4.3 Synchronous versus Asynchronous controllers	77
4.5 Reading Sensor Information	78
4.6 Controlling Actuators	78
4.7 Going further with the Supervisor Controller	79
4.8 Interfacing Webots to third party software	79
4.8.1 Overview	79
4.8.2 Main advantages	79
4.8.3 Limitations	80

4.8.4	MatLab™ TCP/IP utility	80
4.8.5	MatLab™ C interface	81
4.9	Programming Webots controllers with URBI	81
4.9.1	Introduction to URBI	81
4.9.2	Running the examples	82
4.9.3	Kiki and URBI	82
4.9.4	Going further	83
4.9.5	Making your own URBI for Webots controllers	84
4.9.6	Customizing and contributing to URBI for Webots	84
5	Fast2D Simulation Mode	85
5.1	Introduction	85
5.2	Plugin Architecture	85
5.2.1	Overview	85
5.2.2	Dynamically Linked Library	86
5.2.3	Enki Plugin	86
5.3	How to Design a Fast2D Simulation	87
5.3.1	3D to 2D	87
5.3.2	Scene Tree Simplification	88
5.3.3	Bounding Objects	88
5.4	Developing Your Own Fast2D Plugin	88
5.4.1	Header File	88
5.4.2	Fast2D Plugin Types	88
5.4.3	Fast2D Plugin Functions	90
5.4.4	Fast2D Plugin Execution Scheme	94
5.4.5	Fast2D Execution Example	96
6	Using the Khepera™ robot	99
6.1	Hardware configuration	99
6.2	Running the simulation	100
6.3	Understanding the model	102
6.3.1	The 3D scene	102

6.3.2	The Khepera model	103
6.4	Programming the Khepera robot	104
6.4.1	The controller program	104
6.4.2	Looking at the source code	104
6.4.3	Compiling the controller	107
6.5	Transferring to the real robot	107
6.5.1	Remote control	107
6.5.2	Cross-compilation and upload	108
6.6	Working extension turrets	109
6.6.1	The K213 linear vision turret	109
6.6.2	The Gripper turret	109
6.6.3	Custom turrets and Khepera protocol	111
6.7	Support for other K-Team robots	111
6.7.1	Koala™	111
6.7.2	Alice™	111
7	Using the LEGO Mindstorms™ robots	113
7.1	Building up the Rover robot	113
7.2	Webots model of the Rover robot	135
7.3	Transferring to the real Rover robot	136
7.3.1	leJOS	136
7.3.2	Installation	136
7.3.3	Cross-compilation and upload	137
7.3.4	How does it work ?	137
8	Using the Aibo™ robots	139
8.1	Introduction	139
8.2	Hardware configuration	140
8.2.1	Hardware requirements	140
8.2.2	Setting up a wireless link with Aibo	141
8.2.3	Mounting the Memory stick	142
8.2.4	Setting up your Memory stick	143

8.3	Running the simulation	143
8.3.1	Default Aibo world	144
8.3.2	Other Aibo worlds	144
8.4	Understanding the Aibo models	146
8.4.1	Reuseable Aibo objects	146
8.4.2	General model overview	147
8.4.3	Aibo ERS-210 model	149
8.4.4	Aibo ERS-7 model	153
8.5	Using the Aibo control panel	156
8.5.1	Remote (network) functions	163
8.5.2	Manual controls and feedback	164
8.5.3	Motion sequence (MTN) playback	169
8.6	Programming your Aibo	172
8.6.1	Getting started	172
8.6.2	Default controllers	175
8.6.3	Sample controller: MTN Playlist	177
8.6.4	Sample controller: Mimic	180
8.7	Transfer to real robot	182
8.7.1	Cross-compilation	182
9	ALife Contest	185
9.1	Previous Editions	185
9.2	Rules	185
9.2.1	Subject	185
9.2.2	Robot Capabilities	186
9.2.3	Programming Language	187
9.2.4	Scoring Rule	187
9.2.5	Participation	188
9.2.6	Schedule	188
9.2.7	Prize	188
9.3	Web Site	190

9.4	How to Enter the Contest	190
9.4.1	Obtaining the software	190
9.4.2	Running the software	190
9.4.3	Creating your own robot controller	191
9.4.4	Submitting your controller to the ALife contest	192
9.4.5	Analysing the performance and improving your competing controller . . .	193
9.5	Developers' Tips and Tricks	195
9.5.1	Practical issues	195
9.5.2	Java Security Manager	195
9.5.3	Levels of Intelligence	196
10	Robot Soccer Lab	197
10.1	Setup	197
10.2	Rules	198
10.3	Programming	198
10.4	Extensions	199
10.4.1	Modifying the soccer field	199
10.4.2	Modifying the robots	199
10.4.3	Modifying the match supervisor	199

Chapter 1

Installing Webots

1.1 Hardware requirements

Webots is available for Linux i386, Mac OS X and Windows. Other versions of Webots for other UNIX systems (Solaris, Linux PPC, Irix) may be available upon request.

OpenGL hardware acceleration is supported on Windows, Mac OS X and in some Linux configurations. It may also be available on other UNIX systems.

1.2 Installation procedure

To install Webots, you must follow the instructions corresponding to your computer / operating system listed below:

1.2.1 RedHat Linux i386

Webots will run on RedHat Linux distributions, starting from RedHat 9.0. Webots may run on other Linux distributions. For example, it can be easily installed on Debian Linux, using the `alien` command to translate the `rpm` package into a `deb` package before installation. If you do use Red Hat Linux, please refer to your Linux distribution documentation to get the Webots `rpm` package installed.

1. Log on as `root`
2. Insert the Webots CD-ROM, mount it (this might be automatic) and install the following packages

```
mount /mnt/cdrom
cd /mnt/cdrom/linux
rpm -Uvh lib/mjpegtools-1.6.2-1.i386.rpm
# mjpegtools is useful to create MPEG movies from simulations
rpm -Uvh webots/webots-5.0.10-1.i386.rpm
rpm -Uvh webots/webots-kros-1.1.0-1.i386.rpm
# webots-kros is useful only if you want to cross-compile
# controllers for the Khepera robot
```

You may need to use the `--nodeps` or the `--force` if `rpm` fails to install the packages.

1.2.2 Windows XP

1. Uninstall any previous release of Webots or Webots-kros, if any, from the **Start** menu, **Control Panel**, **Add / Remove Programs**. or from the **Start** menu, **Cyberbotics**, **Uninstall Webots** or **Uninstall Webots-kros**.
2. Insert the Webots CD-ROM and open it.
3. Go to the `windows\webots` directory on the CD-ROM.
4. Double click on the `webots-5.0.10_setup.exe` file.
5. Follow the installation instructions.
6. Optionally, double click on the `webots-kros-1.1.0_setup.exe` file to install the cross-compiler for the Khepera robots.

In order to be able to compile controllers, you will need to install a C/C++ development environment. We recommend to use Dev-C++ which is provided on the Webots CD-ROM (in the `windows/utills` directory) as well as from the Bloodshed.net¹ web site. Dev-C++ is an integrated development environment (IDE) for C/C++ with syntax highlighting running on Windows. It includes the MinGW distribution with the GNU GCC compiler and utilities. This software is distributed under the terms of the GNU public license and hence is free of charge.

You may also choose to use Microsoft Visual C++TM if you own a license of this software.

1.2.3 Mac OS X, version 10.3

1. Insert the Webots CD-ROM and open it.
2. Go to the `mac:webots` directory on the CD-ROM.

¹<http://www.bloodshed.net>

3. Double click on the `webots-5.0.10.dmg` file.
4. This will mount on the desktop a volume named `webots` containing the `webots` folder. Move this folder to your applications directory or wherever you would like to install Webots.
5. Set the `WEBOTS_HOME` environment variable to point to the `webots` directory. This is useful to be able to compile the robot controllers using the provided *Makefiles*.

In order to be able to compile controllers, you will need to install the Apple Mac OS X Developer tools, included in the Mac OS X installation CD-ROMs. File editing and compilation using Webots Makefiles can be achieved through these Apple tools. You will probably use the Project Builder application to edit the source codes of the Webots controllers and the Terminal application for invoking `make` from the directory in which your controller gets compiled.

If you would like to be able to create MPEG movies from your Webots simulations, you will have to install the `mjpegtools` package. This package is located in the `linux:lib` directory of the CD-ROM. Install it from the `jpegtools-1.6.2.tar.gz` tar ball. After installation, you should be able to call `mpeg2enc` from the command line.

The CodeWarriorTM development environment is not supported for the development of controllers (although it may also work).

1.3 Registration Procedure

1.3.1 Webots license

Starting with Webots 5, a new license system has been introduced to facilitate the use of Webots.

When installing Webots, you will get a license file, called `webots.key`, containing your name, address, serial number and computer ID. This encrypted file will enable you to use Webots according to the license you purchased. This file is strictly personal: you are not allowed to provide copies of it to any third party in any way, including publishing that file on any Internet server (web, ftp, or any other server). Any copy of your license file is under your responsibility. If a copy of your license file is used by an unauthorized third party to run Webots, then Cyberbotics may engage legal procedures against you. Webots licenses are (1) non-transferable and (2) non-exclusive. This means that (1) you cannot sell or give your Webots license to any third party, and (2) Cyberbotics and its official Webots resellers may sell or give Webots licenses to third parties.

If you need further information about license issues, please send an e-mail to:

`<license@cyberbotics.com>`

Please read your license agreement carefully before registering. This license is provided within the software package. By using the software and documentation, you agree to abide by all the provisions of this license.

1.3.2 Registering

After installing Webots, you will need to register your copy of Webots to get a license file called `webots.key` allowing you to use all the features of Webots corresponding to the license you purchased.

Regular `webots.key` license files are tied to a specific computer, making it impossible to use Webots on another computer. However, if for some reason, you would like to move your Webots license from a computer to another, just send us an e-mail at `<license@cyberbotics.com>` to explain the problem. If you plan to use several Webots licenses over a large number of computers, you should probably ask us to use the floating license server (see below for details). Otherwise, you can jump to the simple registration subsection.

Floating license server: `lserv`

If you purchased a Webots license including a floating license server (either Webots PRO with floating license or Webots EDU), you will be able to install the floating license server for Webots. This software, called `lserv`, allows you to run Webots concurrently on several machines defined by their IP addresses (or their names). Hence Webots is not tied to a predefined number of machines but can be run on an unlimited number of computers. However, the license server takes care that the number of computers running Webots simultaneously doesn't exceed the maximum allowed by the license file. `lserv` should be installed on a server machine, i.e., a computer that is on when users are supposed to run Webots.

Currently, `lserv` only runs on the Linux and Solaris operating systems. However, it allows Webots execution on Linux, Windows and Mac OS X machines. You need to provide Cyberbotics with the MAC address of the `eth0` network card of the server machine running `lserv` so that a special `webots.key` license file can be created and will be sent to you. To know this MAC address on a Linux machine, simply issue `ifconfig eth0` as `root` and read the `HWaddr` parameter. It looks like: `00:50:04:1E:0E:38`. Then, you will need to configure the server and clients to setup the floating license server for your local network.

`lserv` is available for Webots PRO and Webots EDU only upon request to your local Webots reseller.

Please follow the simple registration procedure to provide Cyberbotics with all the information necessary to create the `webots.key` license file for `lserv`. The computer ID provided should be the MAC address of your Linux server on which `lserv` will be running.

Online form

In order to proceed, launch Webots on the computer on which you would like to install the license file. Go the **Register** menu item of the **Help** menu of Webots and follow the instructions. If this computer is connected to the Internet, everything will run smoothly, fill in the requested form

and you will shortly receive the `webots.key` license file via e-mail. Otherwise, you will have to fill in a form² on the website of Cyberbotics). You will then receive an e-mail containing the `webots.key` file corresponding to your license.

Please take care to properly fill in each required field of this form. The *Serial Number* is the serial number of your Webots package which is printed on the CD-ROM under the heading *S/N:* or was communicated to you by e-mail. The `ComputerID` is given by Webots in the **Register** menu item of the **Help** menu.

After completing this form, click on the **Submit** button. You will receive shortly thereafter an e-mail containing your personal license file `webots.key` which is needed to install a registered copy of Webots as described below.

Copying the license file

Once you received it by e-mail, just copy the `webots.key` license file into the `resources` directory of your Webots installation.

Under Linux, copy your personal `webots.key` file into the `/usr/local/webots/resources` directory where Webots was just installed.

Under Mac OS X, copy your personal `webots.key` file into the `Webots:resources` directory where Webots was just installed.

Under Windows, copy your personal `webots.key` file into the directory where Webots was just installed, which is usually `C:\Program Files\Webots\resources`.

²<http://www.cyberbotics.com/registration/webots>

Chapter 2

Getting Started with Webots

To run a simulation in Webots, you need three things:

This chapter gives an overview of the basics of Webots, including the display of the world in the main window and the structure of the `.wbt` file appearing in the scene tree window.

Robot and Supervisor controllers will be explained in detail in chapter 4.

2.1 Introduction to Webots

2.1.1 What is Webots ?

Webots is a professional mobile robot simulation software. It contains a rapid prototyping tool allowing the user to create 3D virtual worlds with physics properties, such as mass repartition, joints, friction coefficients, etc. The user can add simple inert objects or active objects called mobile robots. These robots can have different locomotion schemes (wheeled robots, legged robots or flying robots). Moreover, they be equipped with a number of sensor and actuator devices, like distance sensors, motor wheels, cameras, servos, touch sensors, grippers, emitters, receivers, etc. Finally the user can program each robot individually to exhibit a desired behavior.

Webots contains a large number of robot models and controller program examples that help the users get started.

Webots also contains a number of interfaces to real mobiles robots, so that once your simulated robot behaves as expected, you can transfer its control program to a real robot like Khepera, Hemisson, LEGO Mindstorms, Aibo, etc.

2.1.2 What can I do with Webots ?

Webots is well suited for research and education projects related to mobile robotics. Many mobile robotics projects have been relying on Webots for years in the following areas:

- Mobile robot prototyping (academic research, automotive industry, aeronautics, vacuum cleaner industry, toy industry, hobbyism, etc.)
- Multi-agent research (swarm intelligence, collaborative mobile robots groups, etc.)
- Adaptive behavior research (Genetic evolution, neural networks, adaptive learning, AI, etc.).
- Mobile robotics teaching (robotics lectures, C/C++/Java programming lectures, robotics contest, etc.)

2.1.3 What do I need to use Webots ?

To use Webots, you will need the following hardware:

- A fairly recent PC or Macintosh computer. We recommend at least a Pentium or PowerPC CPU cadenced at 500Mhz. Webots works fine on desktop as well as laptop computers.
- A 3D capable graphics card, with at least 16MB RAM video memory. We recommend nVidia graphics card for PC/Linux users. ATI graphics card are also well suited for Microsoft Windows and Apple Mac OS operating systems.

The following operating system are supported:

- Linux. Although only RedHat Linux is officially supported, Webots is known to run on most major Linux distribution, including Mandrake, Debian, SuSE, Slackware, etc. We recommend however to use a fairly recent recent version of Linux. Webots is provided as an RPM package, as well as a DEB package.
- Windows. Webots runs under Windows 2000, XP and 2003. It doesn't run on Windows 98, ME or NT4.
- Mac OS X. Version 10.3 of Mac OS X or ealier is highly recommended, as Webots hasn't been tested on older versions of Mac OS X and may not work as expected on such old versions.

Usually, you will need to be administrator to be able to install Webots. Once installed, Webots can be used as a regular unprivileged user.

Although no special knowledge is needed to simply view the demos of robot simulations in Webots, you will need a minimal amount of scientific and technical knowledge to be able to develop your own simulations:

- A basic knowledge of C, C++ or Java programming languages is necessary to program your own robot controllers efficiently. However, even if you don't know these languages, you can still program the Hemisson robot using a simple graphical programming language called BotStudio.
- If you don't want to use existing robot models provided within Webots and would like to create your own robot models, or add special objects in the simulated environments, you will need some very basic knowledge of 3D computer graphics and VRML97 3D description language. That will allow you to create 3D models in Webots or import them from a 3D modelling software.

2.1.4 What is a world ?

A world in Webots is a 3D virtual environment in which you can create objects and robots. A world is saved in the `worlds` directory, in a `.wbt` file which contains a description for any object: Its position, orientation, geometry, appearance (like color, brightness), physical properties, type of object, etc. A world is a hierarchical structure where objects can contain other objects (like in VRML97). For example a robot can contain two wheels, a distance sensor and a servo which itself contains a camera, thus making the camera moveable relatively to the robot thanks to the servo. However, a world file does not contain all the information necessary to run a simulation. The controller of each robot is specified in the world file by a reference to an executable binary file, but the world file doesn't contain this executable binary file.

2.1.5 What is a controller ?

A controller is an executable binary file which is used to control a robot described in a world file. Controllers are stored in subdirectories of the Webots `controllers` directory. Controllers may be native executables files (`.exe` under Windows) or Java binary files (`.class`).

2.2 Launching Webots

2.2.1 On Linux

From an X terminal, type `webots` to launch the simulator. You should see the world window appear on the screen (see figure 2.1).

Webots can also run in batch mode, that is without displaying any window. This is useful to launch simulations from scripts to perform extensive simulations with different sets of parameters and save results automatically from a supervisor or robot controller process. To launch Webots in batch mode, simply type `webots --batch filename.wbt` where `filename.wbt` is

the name of the world file you want to use. Webots will then be launched in batch mode: The speed of execution should correspond to the fast mode.

2.2.2 On Mac OS X

Open the directory in which you uncompressed the Webots package and double-click on the Webots icon. You should see the world window appear on the screen (see figure 2.1).

2.2.3 On Windows

From the **Start** menu, go to the **Program Files — Cyberbotics** menu and click on the **Webots 5.0.10** menu item. You should see the world window appear on the screen (see figure 2.1).

2.3 Main Window: Menus and buttons

The main window allows you to display your virtual worlds and robots described in the `.wbt` file. Four menus and a number of buttons are available.

2.3.1 File menu and shortcuts

The **New** menu item opens a new default world representing a chessboard of 10 x 10 plates on a surface of 1 m x 1 m. The following button can be used as a shortcut:



The **File** menu will also allow you to perform the standard file operations: **Open...**, **Save** and **Save As...**, respectively, to load, save and save with a new name the current world.

The following buttons can be used as shortcuts:



The **Revert** item allows you to reload the most recently saved version of your `.wbt` file.

The following button can be used as a shortcut:





Figure 2.1: Webots main window

The **Export VRML 2.0...** item allows you to save the `.wbt` file as a `.wrl` file, conforming to the VRML97 standard. Such a file can, in turn, be opened with any VRML97 viewer. This is especially useful for publishing a world created with Webots on the Web.

The **Make Animation...** item allows you to create a 3D animation as a WVA file. This file format is useful to playback Webots animations in real 3D, including navigation facilities. The WVA viewer is called Webview. It is a freely available software downloadable from Cyberbotics' Webview web site¹. It can run as a plugin for Internet Explorer, Netscape or Mozilla, but also as a stand alone application. Webview works on Windows, Linux and Mac OS X. It is well suited to demonstrate Webots results, possibly on the Internet World Wide Web.

The **Make Movie...** item allows you to create a MPEG movie under Linux and Mac OS X or an AVI movie under Windows. As movies are created on a 25 frame per second basis, you should

¹<http://www.cyberbotics.com/webview>

adapt the `basicTimeStep` and the `displayRefresh` fields of the `WorldInfo` node in the scene tree window to obtain a movie running at real time. Leaving the basic time step to 32 ms and setting the display refresh each 1 basic simulation step should produce movies running slightly faster than real time. If you need exact real time, set the basic time step to 25 ms (it might then be optimal to adapt your controllers' `robot_step` functions using a multiple value of 25, like 50, 75 or 100). It is also possible to make accelerated movies by setting the display refresh each 2 (or more) basic time step while leaving the basic time step to its original value (32 or 25).

The **Screenshot...** item allows you to take a screenshot of the current view in Webots. It opens a file dialog to save the current view as a PNG image.

2.3.2 Edit menu

The **Scene Tree Window** menu item opens the window in which you can edit the world and the robot(s). A shortcut is available by double-clicking on a solid in the world. A solid is a physical object in the world.

The **Import VRML 2.0...** menu item inserts VRML97 objects at the end of the scene tree. These objects come from a VRML97 file you will have to specify. This feature is useful to import complex shapes that were modeled in a 3D modelling software, then exported to VRML97 (or VRML 2.0), and then imported into Webots with this menu item. Most 3D modelling software, like 3D Studio Max, Maya, AutoCAD, Pro Engineer, AC3D or Art Of Illusion, include the VRML97 (or VRML 2.0) export feature. Beware, Webots cannot import VRML 1.0 file format. Once imported, these objects appear as `Group`, `Transform` or `Shape` nodes at the bottom of the scene tree. You can then either turn these objects into Webots nodes (like `Solid`, `DifferentialWheels`, etc.) or cut and paste them into the `children` list of existing Webots nodes.

The **Restore Viewpoint** menu item resets the camera position and orientation as it was originally when the file was open. This feature is handy when you get lost while navigating in the scene and want to return to the original camera position and orientation.

The **Preferences** item pops up a window with the following panels:

- **General:** The `Startup` mode allows you to choose the state of the simulation when Webots is launched (stop, run, fast; see the **Simulation** menu).
- **Rendering:** This tab controls the 3D rendering in the simulation window.
 - Checking the `Display axes` check box displays a red, green and blue axes representing respectively the x, y and z axes of the world coordinate system.
 - Checking the `Display sensor rays` check box displays the distance sensor rays of the robot(s) as red lines.
 - Checking the `Display lights` check box displays the lights (`PointLight` in the world so that they can be moved more accurately).

- **Files and paths:** The user directory and the default `.wbt` world which is open when launching Webots are defined here. The user directory should contain at least a `worlds`, `controllers`, `physics`, and `objects` subdirectories where Webots will be looking for files. A complete user directory can be created easily from the **Setup user directory** menu item in the **Wizard** menu

2.3.3 Simulation menu and the simulation buttons

In order to run a simulation a number of buttons are available corresponding to menu items found under the **Simulation** menu:



Stop: Interrupt **Run** or **Fast** modes.



Step: Execute one basic time step of simulation. The duration of such a step is defined in the `basicTimeStep` field of the `WorldInfo` node and can be adjusted in the scene tree window to suit your needs.



Run: Execute simulation steps until the **Stop** mode is entered. In run mode, the 3D display of the scene is refreshed every `n` basic time step, where `n` is defined in the `displayRefresh` field of the `WorldInfo` node.



Fast: Same as **Run**, except that no display is performed (Webots PRO only).

The **Fast** mode performs a very fast simulation mode suited for heavy computation (genetic algorithms, vision, learning, etc.). However, as the world display is disabled during a **Fast** simulation, the scene in the world window remains blank until the **Fast** mode is stopped. This feature is available only with Webots PRO.

The **World View / Robot View** item allows you to switch between two different points of view:

- **World View:** This view corresponds to a fixed camera standing in the world.
- **Robot View:** This view corresponds to a mobile camera following a robot.

The default view is the world view. If you want to switch to the **Robot View**, first select the robot you want to follow (click on the pointer button then on the robot), and then choose **Robot View** in the **Simulation** menu. To return to the **World View** mode, reselect this item.

A speedometer (see figure 2.2) allows you to observe the speed of the simulation on your computer. It is displayed in the bottom right hand side of the main window and indicates how fast the simulation runs compared to real time. In other words, it represents the speed of the virtual time. If the value of the speedometer is 2, it means that your computer simulation is running twice as



Figure 2.2: Speedometer

fast as the corresponding real robots would. This information is relevant both in **Run** mode and **Fast** mode.

To the left of the speedometer, the virtual time is displayed using the following format: *H:MM:SS:MMM* where *H* is the number of hours (may lie on several digits), *MM* is the number of minutes, *SS* is the number of seconds and *MMM* is the number of milliseconds. (see figure 2.2). If the speedometer value is higher than one, the virtual time will be progressing faster than the real time. This information is relevant both in **Run** mode and **Fast** mode.

The basic time step for simulation can be set in the `basicsTimeStep` field of the `WorldInfo` node in the scene tree window. It is expressed in virtual time milliseconds. The value of this time step defines the duration of the time step executed during the **Step** mode. This step is multiplied by the `displayRefresh` field of the same `WorldInfo` node to define how frequently the display is refreshed.

In **Run** mode, with a time step of 64 ms and a fairly simple world displayed with the default window size, the speedometer will typically indicate approximately 0.5 on a Pentium II / 266 Mhz without hardware acceleration and 12 on a Pentium III / 500 Mhz with an nVidia Geforce II MX graphics card.

2.3.4 Wizard menu

The **Wizard** menu is useful to facilitate the creation of a new user directory (from the **Setup user directory** menu item) or the creation of a new robot controller (from the **New robot controller** menu item).

The **Setup user directory** menu item will ask you to choose a name for your user directory. A user directory is a directory that will contain all the files you will create while using Webots, including world file, controller files, object files, physics shared libraries, etc. Once you chose a name for this user directory, you will be asked to choose a location on your hard disk where to store it. Then, Webots will create this directory at the specified location and it will create all the subdirectories and files needed. Finally, it will set this directory as the current user directory in the Webots preferences. From there, you will be able to save all your Webots files in the subdirectories of this user directory (world files, controllers, etc.).

The **New robot controller** menu item allows you to create a new controller program for your robot. You will be prompted to choose between a C, C++ or a Java controller. Then, Webots will ask

you the name of your controller and it will create all the necessary files (including a template source code file) in your user directory.

2.3.5 Help menu

In the **Help** menu, the **About...** item opens the `About . . .` window, displaying the license information.

The **Introduction** item is a short introduction to Webots (HTML file). You can access the User Guide and the Reference Manual with the **User Guide** and **Reference Manual** items (PDF files). The **Web site of Cyberbotics** item lets you visit our Web site.

2.3.6 Navigation in the scene

The view of the scene is generated by a virtual camera set in a given position and orientation. You can change this position and orientation to navigate in the scene using the mouse buttons. The x , y , z axes mentioned below correspond to the coordinate system of the camera; z is the axis corresponding to the direction of the camera.

- *Rotate viewpoint:* To rotate the camera around the x and y axis, you have to set the mouse pointer in the 3D scene, press the left mouse button and drag the mouse:
 - if you clicked on a solid object, the rotation will be centered around the origin of the local coordinate system of this object.
 - if you clicked outside of any solid object, the rotation will be centered around the origin of the world coordinate system.
- *Translate viewpoint:* To translate the camera in the x and y directions, you have to set the mouse pointer in the 3D scene, press the right mouse button and drag the mouse.
- *Zoom / Tilt viewpoint:* Set the mouse pointer in the 3D scene, then:
 - if you press both left and right mouse buttons (or the middle button) and drag the mouse vertically, the camera will zoom in or out.
 - if you press both left and right mouse buttons (or the middle button) and drag the mouse horizontally, the camera will rotate around its z axis (tilt movement).
 - if you use the wheel of the mouse, the camera will zoom in or out.

2.3.7 Moving a solid object

In order to move an object, hold the shift key down while using the mouse.

- Translation: Pressing the left mouse button while the shift key is pressed allows you to drag solid objects on the ground (xz plan).
- Rotation: Pressing the right mouse button while the shift key is pressed rotates solid objects: A first click is necessary to select a solid object, then a second shift-press-and-drag rotates the selected object around its y axis.
- Lift: Pressing both left and right mouse buttons, the middle mouse button, or rolling the mouse wheel while the shift key is pressed allows you to lift up or down the selected solid object.

2.3.8 Selecting a solid object

Simply clicking on a solid object allows you to select this object. Selecting a robot enables the choice of **Robot View** in the **simulation** menu. Double-clicking on a solid object opens the scene tree window where the world and robots can be edited. The selected solid object appears selected in the scene tree window as well.

2.4 Scene Tree Window

As seen in the previous section, to access to the Scene Tree Window you can either choose **Scene Tree Window** in the **Edit** menu, or click on the pointer button and double-click on a solid object.

The scene tree contains all information necessary to describe the graphic representation and simulation of the 3D world. A world in Webots includes one or more robots and their environment.

The scene tree of Webots is structured like a VRML97 file. It is composed of a list of nodes, each containing fields. Fields can contain values (text string, numerical values) or nodes.

Some nodes in Webots are VRML97 nodes, partially or totally implemented, while others are specific to Webots. For instance the `Solid` node inherits from the `Transform` node of VRML97 and can be selected and moved with the buttons in the World Window.

This section describes the buttons of the Scene Tree Window, the VRML97 nodes, the Webots specific nodes and how to write a `.wbt` file in a text editor.

2.4.1 Buttons of the Scene Tree Window

The scene tree with the list of nodes appears on the left side of the window. Clicking on the `+` in front of a node or double-clicking on the node displays the fields inside the node, and similarly expands the fields. The field values can be defined on the top right side of the window. Five editing buttons are available on the bottom right side of the window:

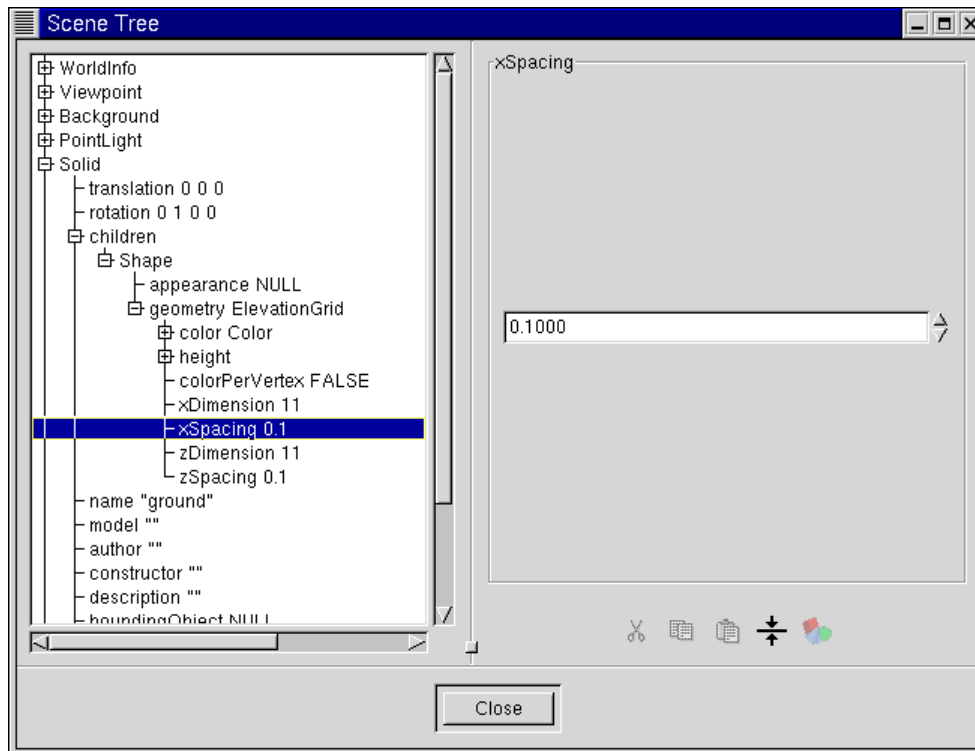


Figure 2.3: Scene Tree Window



Cut



Copy



Paste after

These three buttons let you cut, copy and paste nodes and fields. However, you can't perform these operations on the first three nodes of the tree (*WorldInfo*, *Viewpoint* and *Background*). These nodes are mandatory and cannot be duplicated. Similarly, you can't copy the *Supervisor* node because only one supervisor is allowed. Please note that when you cut or copy a robot node, like a *DifferentialWheels* or *Supervisor* node, the *controller* field of this node is reset to "void".



Delete: This button allows you to delete a node. It appears only if a node is selected. If a field is selected, the **Default Value** button appears instead.



Default Value: You can click on this button to reset a field to its default value(s). A field with values must be selected in order to perform this button. If a node is selected, the **Delete** button replaces it.



Transform: This button allows you to transform a node into another one.



Insert after: With this button, you can insert a node after the one currently selected. This new node contains fields with default values, which you can of course modify to suit your needs. This button also allows you to add a node to a `children` field. In all cases, the software only permits you to insert a coherent node.



Insert Node: Use this to insert a node into a field whose value is a node. You can insert only a coherent node.



Export Node: Use this button to export a node into a file. Usually, nodes are saved in your `objects` directory. Such saved nodes can then be reused in other worlds.



Import Node: Use this button to import a previously saved node into the scene tree. Usually, saved nodes are located in the Webots `objects` directory or in your own `objects` directory. The Webots `objects` directory already contains a few nodes that can be easily imported.

2.4.2 VRML97 nodes

A number of VRML97 nodes are partially or completely supported in Webots.

The exact features of VRML97 are the subject of a standard managed by the International Standards Organization (ISO/IEC 14772-1:1997).

You can find the complete specifications of VRML97 on the official VRML97 Web site².

The following VRML97 nodes are supported in Webots:

- Appearance
- Background
- Box
- Color
- Cone
- Coordinate
- Cylinder

²<http://www.web3d.org>

- DirectionalLight
- ElevationGrid
- Fog
- Group
- ImageTexture
- IndexedFaceSet
- IndexedLineSet
- Material
- PointLight
- Shape
- Sphere
- Switch
- TextureCoordinate
- TextureTransform
- Transform
- Viewpoint
- WorldInfo

The Webots Reference Manual gives a list of nodes supported in Webots and specify which fields are actually used. For a comprehensive description of the VRML97 nodes, please refer to the VRML97 documentation.

2.4.3 Webots specific nodes

In order to implement powerful simulations including mobile robots with different propulsion schemes (wheeled robots, legged robots or flying robots), a number of nodes specific to Webots have been added to the VRML97 set of nodes.

VRML97 uses a hierarchical structure for nodes. For example, the Transform node inherits from the Group node, such that, like the Group node, the Transform node has a children field, but it also adds three additional fields: translation, rotation and scale.

In the same way, Webots introduces new nodes which inherit from the VRML97 Transform node, principally the Solid node. Other Webots nodes (DifferentialWheels, Camera, TouchSensor, etc.) inherit from this Solid node.

The Reference Manual gives a complete description of all Webots nodes and their respective fields.

2.4.4 Principle of the collision detection

The collision detection engine is able to detect a collision between two Solid nodes. It calculates the intersection between the bounding objects of the solids. A bounding object (described in the boundingObject field of the Solid node) is a geometric shape or a group of geometric shapes which bounds the solid. If the boundingObject field is NULL, then no collision detection is performed for this Solid node. A Solid node may contain other Solid nodes as children, each of them having its own bounding object.

The collision detection is mainly used to detect if a robot (for example a DifferentialWheels node) collides with an obstacle (Solid node), or with another robot. Two Solid nodes can never inter-penetrate each other; their movement is stopped just before the collision.

Example: A solid with a bounding box different from its list of children.

Let us consider the Khepera robot model. It is not exactly a Solid node, but the principle for the boundingObject is the same. Open the khepera.wbt file and look at the boundingObject field of the DifferentialWheels node. The bounding object is a cylinder which has been transformed. See figure 2.4.

2.4.5 Writing a Webots file in a text editor

It is possible to write a Webots world file (.wbt) using a text editor. A world file contains a header, nodes containing fields and values. Note that only a few VRML97 nodes are implemented, and that there are nodes specific to Webots. Moreover, comments can only be written in the DEF, and not like in a VRML97 file.

The Webots header is:

```
#VRML_SIM V4.0 utf8
```

After this header, you can directly write your nodes. The three nodes WorldInfo, Viewpoint and Background are mandatory.

Note: We recommend that you write your file using the tree editor. However it may be easier to make some particular modifications using a text editor (like using the search and replace feature of a text editor).

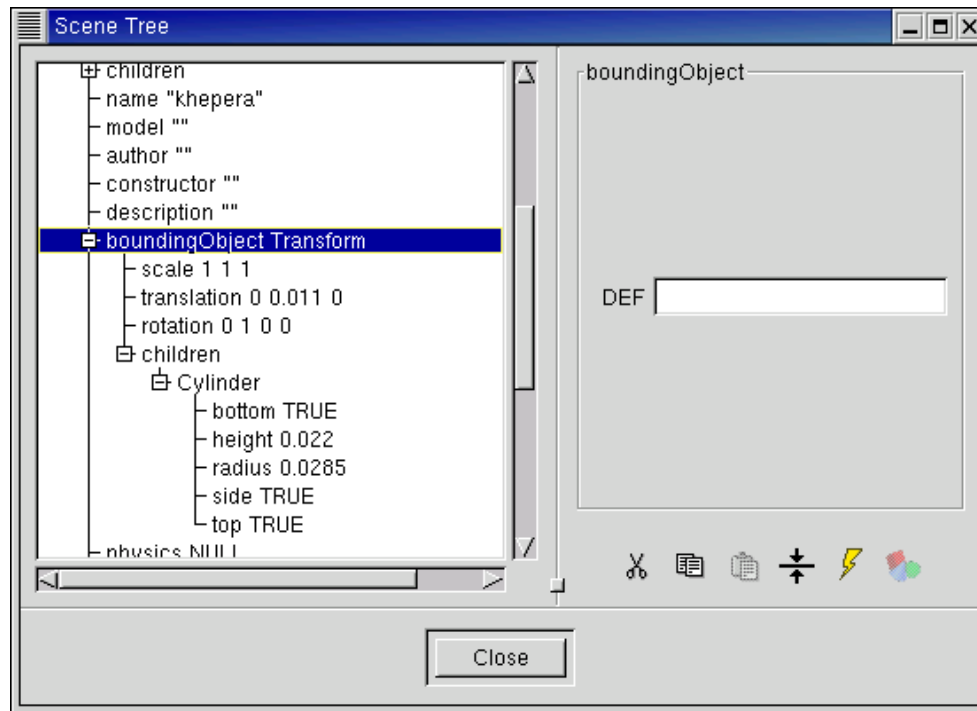


Figure 2.4: The bounding box of the Khepera robot

2.5 Citing Webots

When writing a scientific paper, or describing your project involving Webots on a web site, it is always appreciated to make a correct reference to Webots, mentioning Cyberbotics' web site explicitly and a reference journal paper describing Webots. In order to help you in such a task, we provide here some citation examples, including BibTeX entries that you can freely reuse in your own documents:

2.5.1 Citing Cyberbotics' web site

This project uses Webots³, a commercial mobile robot simulation software developed by Cyberbotics Ltd.

This project uses Webots (<http://www.cyberbotics.com>), a commercial mobile robot simulation software developed by Cyberbotics Ltd.

The BibTeX reference entry may look odd, as it is very different from a standard paper citation and we want the specified fields to appear in the normal plain citation mode of LaTeX.

@MISC{Webots,

³<http://www.cyberbotics.com>

```

AUTHOR = {Webots},
TITLE  = {http://www.cyberbotics.com},
NOTE   = {Commercial Mobile Robot Simulation Software},
EDITOR = {Cyberbotics Ltd.},
URL    = {http://www.cyberbotics.com}
}

```

Once compiled with LaTeX, it should display as follow:

References

[1] *Webots. <http://www.cyberbotics.com>. Commercial Mobile Robot Simulation Software.*

2.5.2 Citing a reference journal paper about Webots

A reference paper was published in the International Journal of Advanced Robotics Systems. Here is the BibTeX entry:

```

@ARTICLE{Webots04,
  AUTHOR   = {Michel, O.},
  TITLE    = {Webots: Professional Mobile Robot Simulation},
  JOURNAL  = {Journal of Advanced Robotics Systems},
  YEAR    = {2004},
  VOLUME  = {1},
  NUMBER  = {1},
  PAGES   = {39--42},
  URL     = {http://www.ars-journal.com/ars/SubscriberArea/Volume1/39-42.pdf}
}

```

Chapter 3

Tutorial: Modeling and simulating your robot

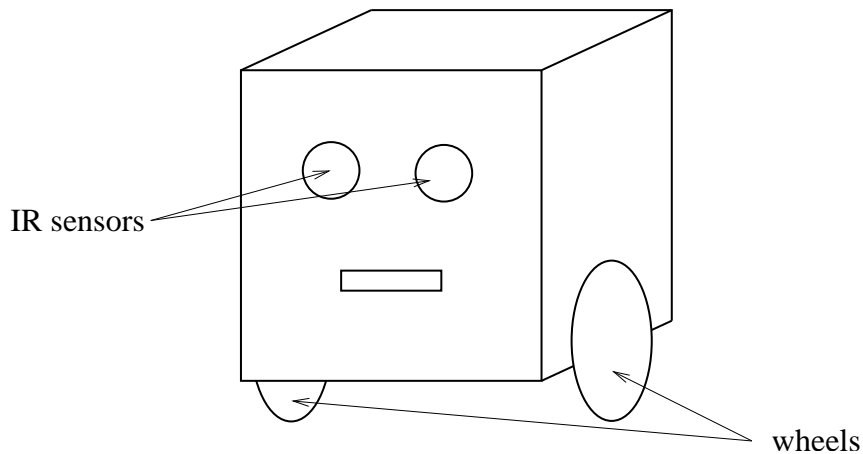
The aim of this chapter is to give you several examples of robots, worlds and controllers. The first world is very simple, nevertheless it introduces the construction of any basic robot, and explains how to program a controller. The second example shows you how to model and use a camera on this simple robot. The third example will add physics to the robot and world, so that the robot can play with a ball. Finally, the last example will show you how to build a virtual Pioneer 2TM robot from ActivMedia Robotics.

3.1 My first world: kiki.wbt

As a first introduction, we are going to simulate a very simple robot made up of a box, two wheels and two infra-red sensors (see figure 3.1). The robot is controlled by a program performing obstacle avoidance inspired from Braitenberg's algorithm. It evolves in a simple environment surrounded by a wall.

3.1.1 Setup

Before starting, please check that Webots was installed properly on your computer (refer to the installation chapter of this manual). Then, you will have to setup a working directory that will contain the files you will create in this tutorial. To do so, create a directory called `my_webots` in your local directory. Then, create a couple of subdirectories called `worlds` and `controllers`. The first one will contain the simulation worlds you will create, while the second one will contain your programs controlling the simulated robots. If you are using `gcc` as a compiler, you may also need to copy the `Makefile.include` file from the Webots `controllers` directory in your local `controllers` directory. To start up with this tutorial, simply copy the `kiki.wbt` worlds from the Webots `worlds` directory to your local `worlds` directory. You will also have

Figure 3.1: The *kiki* robot

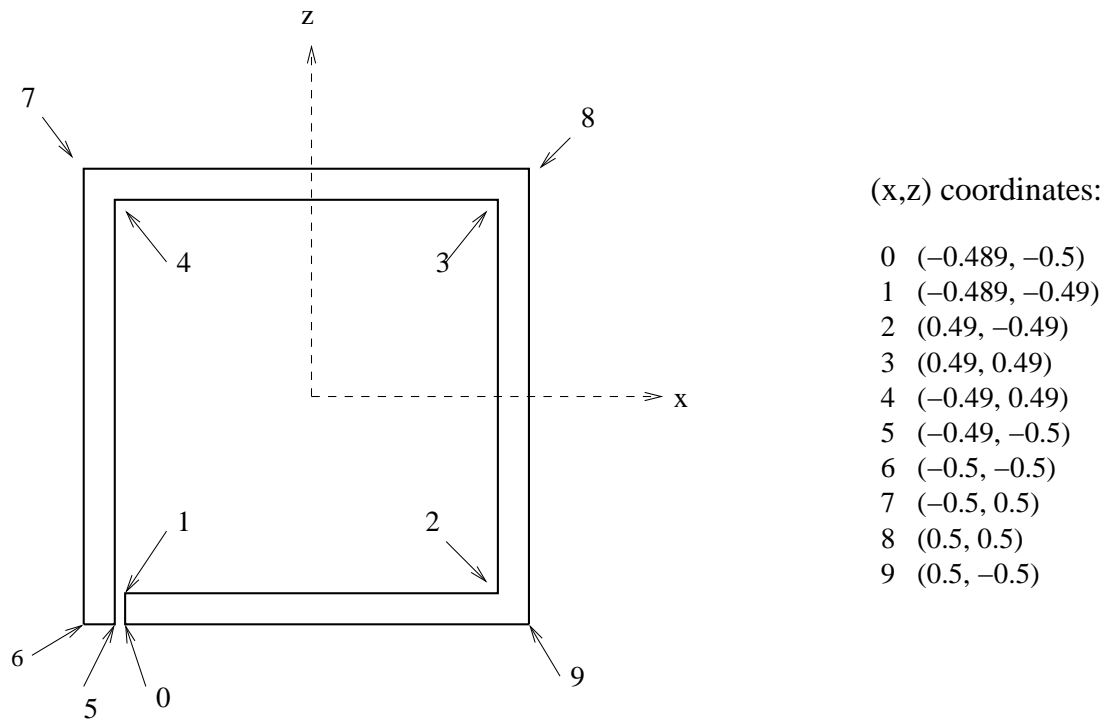
to copy the `kiki` subdirectory which contains the `kiki.png` image. Finally, copy the `simple` directory from the `Webots controllers` directory to your local `controllers` directory. Now you should inform Webots that your working directory is there. To do it, launch Webots and open the **Preferences...** from the **Edit** menu. Select the **Files and paths** tab and select your local `my_webots` directory as the **User directory**. You can also set the `kiki.wbt` world as the default world. Then quit Webots, so that the preferences are saved. When you will restart it, it will run the `kiki` world.

3.1.2 Environment

This very first simulated world is as simple as possible. It includes a floor and a surrounding wall to avoid that the robot escapes. This wall is modelled using an `Extrusion` node. The coordinates of the wall are shown in figure 3.2.

First, launch Webots and stop the current running simulation by pressing the **Stop** button. Go to the **File** menu, **New** item to create a new world. This can also be achieved through the **New** button, or the keyboard shortcut indicated in the **File** menu. Then open the scene tree window from the **Scene Tree...** item in the **Edit** menu. This can also be achieved by double-clicking in the 3D world. Let us start by changing the lighting of the scene:

1. Select the `PointLight` node, and click on the `+` just in front of it. You can now see the different fields of the `PointLight` node. Select `ambientIntensity` and enter 0.6 as a value, then select `intensity` and enter 0.8, then select `location` and enter 0.5 0.5 0.5 as values. Press `return`.
2. Select the `PointLight` node, copy and paste it. Open this new `PointLight` node and type -0.5 0.5 0.5 in the `location` field.

Figure 3.2: The *kiki* world

3. Repeat this paste operation twice again with `-0.5 0.5 -0.5` in the `location` field of the third `PointLight` node, and `0.5 0.5 -0.5` in the `location` field of the fourth and last `PointLight` node.
4. The scene is now better lit. Open the **Preferences...** from the **Edit** menu, select the **Rendering** tab and check the **Display lights** option. Click on the **OK** button to leave the preferences and check that the light sources are now visible in the scene. Try the different mouse buttons, including the mouse wheel if any, and drag the mouse in the scene to navigate and observe the location of the light sources. If you need more explanations with the 3D navigation in the world, go to the **Help** menu and select the **How do I navigate in 3D ?** item.

Secondly, let us create the wall:

1. Select the last `Transform` node in the scene tree window (which is the floor) and click on the **insert after** button.
2. Choose a `Solid` node.
3. Open this newly created `Solid` node from the `+` sign and type "wall" in its name field.
4. Select the children field and **Insert after** a `Shape` node.

5. Open this Shape, select its appearance field and create an Appearance node from the **New node** button. Use the same technique to create a Material node in the material field of the Appearance node. Select the diffuseColor field of the Material node and choose a color to define the color of the wall. Let us make it dark green.
6. Now create an Extrusion node in the geometry field of the Shape.
7. Set the convex field to FALSE. Then, set the wall corner coordinates in the crossSection field as shown in figure 3.2. You will have to re-enter the first point (0) at the last position (10) to complete the last face of the extrusion.
8. In the spine field, write that the wall ranges between 0 and 0.1 along the Y axis (instead of the 0 and 1 default values).
9. As we want to prevent our robot to pass through the walls like a ghost, we have to define the boundingObject field of the wall. Bounding objects cannot use complex geometry objects. They are limited to box, cylinder and spheres primitives. Hence, we will have to create four boxes (representing the four walls) to define the bounding object of the surrounding wall. Select the boundingObject field of the wall and create a Group node that will contain the four walls. In this Group, insert a Transform node as a children. Create a Shape as the unique children of the Transform. Instead of creating a new Appearance for this Shape, reuse the first Appearance you created (for the wall). To do so, go back to the children list of the wall Solid, open the Shape, click on the Appearance node and you will see on the right hand side of the window that you can enter a DEF name. Write WALL_APPEARANCE as a DEF name and return to the Shape of the bounding object. Select its appearance field and create a **New node** for it. However, in the **Create a new node** dialog, you will now be able to use the WALL_APPEARANCE you just defined. Select this item and click **OK**. Now create a Box as a geometry for this Shape node. Set the size of the Box to [1 0.1 0.01], so that it matches the size of a wall. Set the translation field of the Transform node to [0 0.05 0.495], so that it matches the position of a wall. Now, close this Transform, copy and paste it as the second children of the list. Set the translation field of the new node to [0 0.05 -0.495], so that it matches the opposite wall. Repeat this operation with the two remaining walls and set their rotation fields to [0 1 0 1.57] so that they match the orientation of the corresponding walls. You also have to edit their translation field as well, so that they match the position of the corresponding walls.
10. Close the tree editor, save your file as "my_kiki.wbt" and look at the result.

The wall in the tree editor is represented in figure 3.3, while the same wall in the world editor is visible in figure 3.4

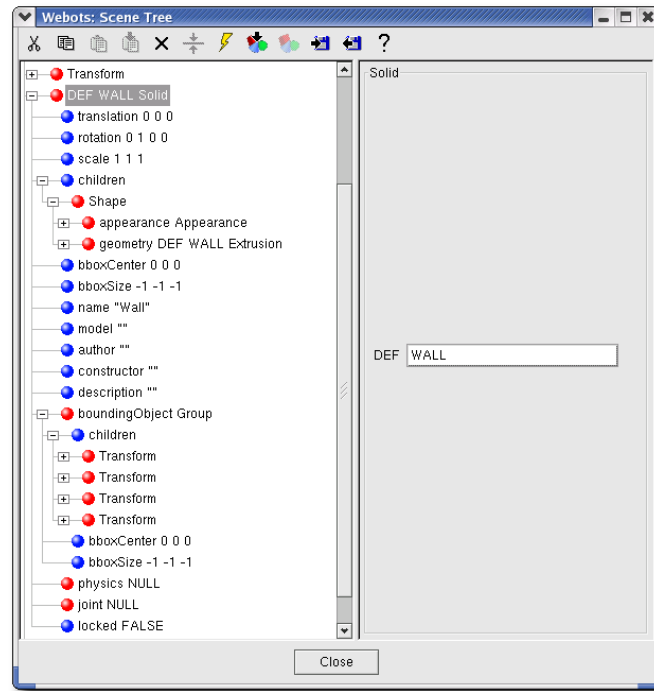


Figure 3.3: The wall in the tree editor

3.1.3 Robot

This subsection describes how to model the *kiki* robot as a `DifferentialWheels` node containing several children: a `Transform` node for the body, two `Solid` nodes for the wheels, two `DistanceSensor` nodes for the infra-red sensors and a `Shape` node with a texture.

The origin and the axis of the coordinate system of the robot and its dimensions are shown in figure 3.5.

To model the body of the robot:

1. Open the scene tree window.
2. Select the last `Solid` node.
3. **Insert after** a `DifferentialWheels` node, set its name to "kiki".
4. In the children field, first introduce a `Transform` node that will contain a shape with a box. In the new children field, **insert after** a `Shape` node. Choose a color, as described previously. In the geometry field, **insert** a `Box` node. Set the size of the box to [0.08 0.08 0.08]. Now set the translation values to [0 0.06 0] in the `Transform` node (see figure 3.6)

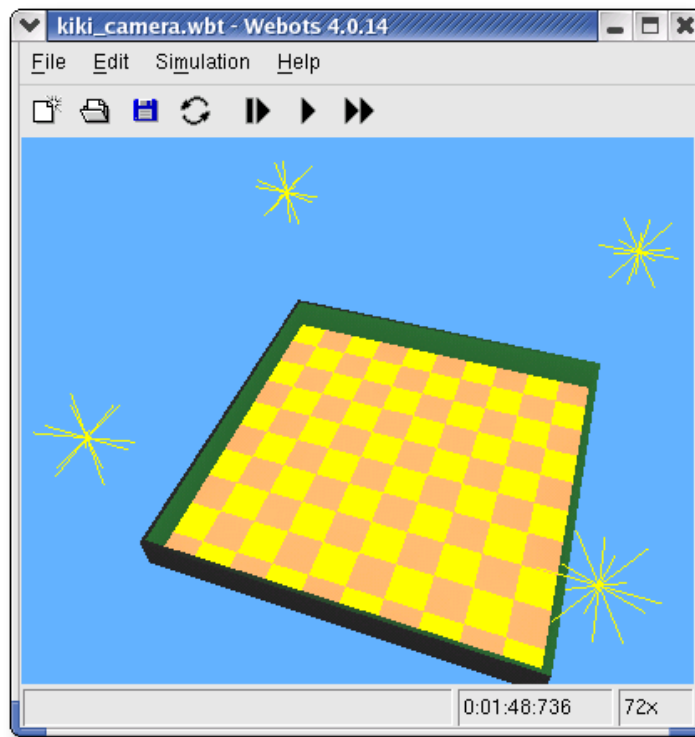


Figure 3.4: The wall in the world window

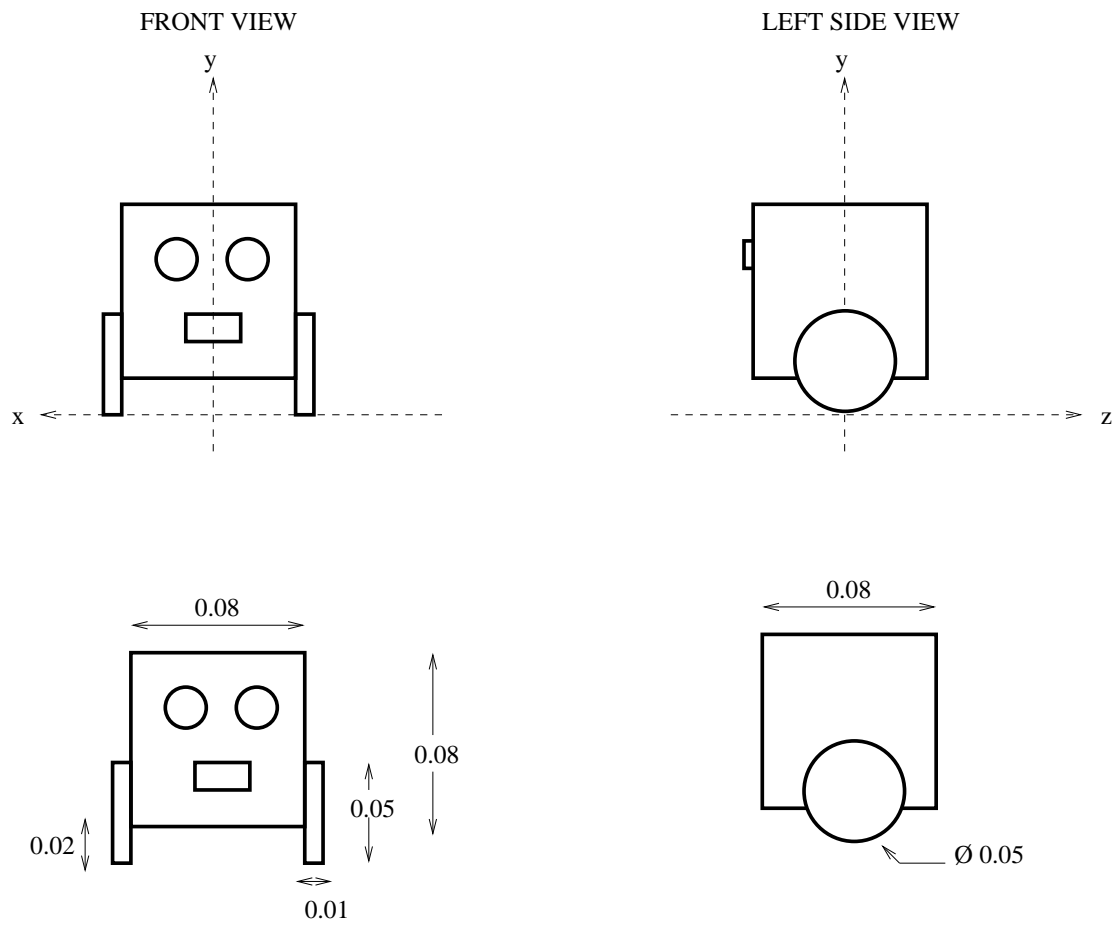


Figure 3.5: Coordinate system and dimensions of the *kiki* robot

To model the left wheel of the robot:

1. Select the `Transform` node corresponding to the body of the robot and **insert after** a `Solid` node in order to model the left wheel. Type "left wheel" in the name field, so that this `Solid` node is recognized as the left wheel of the robot and will rotate according to the motor command.
2. The axis of rotation of the wheel is x . The wheel will be made of a `Cylinder` rotated of $\pi/2$ radians around the z axis. To obtain proper movement of the wheel, you must pay attention not to confuse these two rotations. Consequently, you must add a `Transform` node to the children of the `Solid` node.
3. After adding this `Transform` node, introduce inside it a `Shape` with a `Cylinder` in its geometry field. Don't forget to set an appearance as explained previously. The dimensions of the cylinder should be 0.01 for the height and 0.025 for the radius. Set the rotation to `[0 0 1 1.57]`. Pay attention to the sign of the rotation; if it is wrong, the wheel will turn in the wrong direction.
4. In the `Solid` node, set the translation to `[-0.045 0.025 0]` to position the left wheel, and set the rotation of the wheel around the x axis: `[1 0 0 0]`.
5. Give a DEF name to your `Transform`: `WHEEL`; notice that you positioned the wheel in translation at the level of the `Solid` node, so that you can reuse the `WHEEL` `Transform` for the right wheel.
6. Close the tree window, look at the world and save it. Use the navigation buttons to change the point of view.

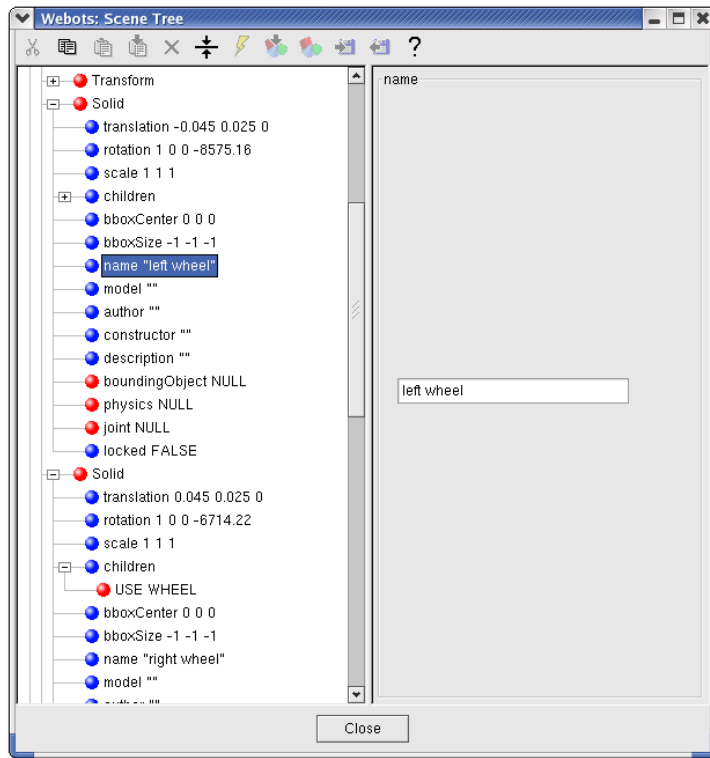
To model the right wheel of the robot:

1. Select the left wheel `Solid` node and **insert after** another `Solid` node. Type "right wheel" in the name field. Set the translation to `[0.045 0.025 0]` and the rotation to `[1 0 0 0]`.
2. In the children, **insert after** `USE WHEEL`. Press `Return`, close the tree window and save the file. You can examine your robot in the world editor, move it and zoom in on it.

The robot and its two wheels are shown in figure 3.7 and figure 3.8.

The two infra-red sensors are defined as two cylinders on the front of the robot body. Their diameter is 0.016 m and their height is 0.004 m. You must position these sensors properly so that the sensor rays point in the right direction, toward the front of the robot.

1. In the children of the `DifferentialWheels` node, **insert after** a `DistanceSensor` node.
2. Type the name "ir0". It will be used by the controller program.

Figure 3.7: Wheels of the *kiki* robot

3. Let's attach a cylinder shape to this sensor: In the children list of the `DistanceSensor` node, **insert after** a `Transform` node. Give a DEF name to it: `INFRARED`, which you will use for the second IR sensor.
4. In the children of the `Transform` node, **insert after** a `Shape` node. Define an appearance and **insert** a `Cylinder` in the `geometry` field. Type 0.004 for the height and 0.008 for the radius.
5. Set the rotation for the `Transform` node to `[0 0 1 1.57]` to adjust the orientation of the cylinder.
6. In the `DistanceSensor` node, set the translation to position the sensor and its ray: `[-0.02 0.08 -0.042]`. In the **File** menu, **Preferences, Rendering**, check the **Display sensor rays** box. In order to have the ray directed toward the front of the robot, you must set the rotation to `[0 1 0 1.57]`.
7. In the `DistanceSensor` node, you must introduce some values of distance measurements of the sensors to the `lookupTable` field, according to figure 3.9. These values are:

```
lookupTable [ 0      1024  0,
             0.05  1024  0,
```

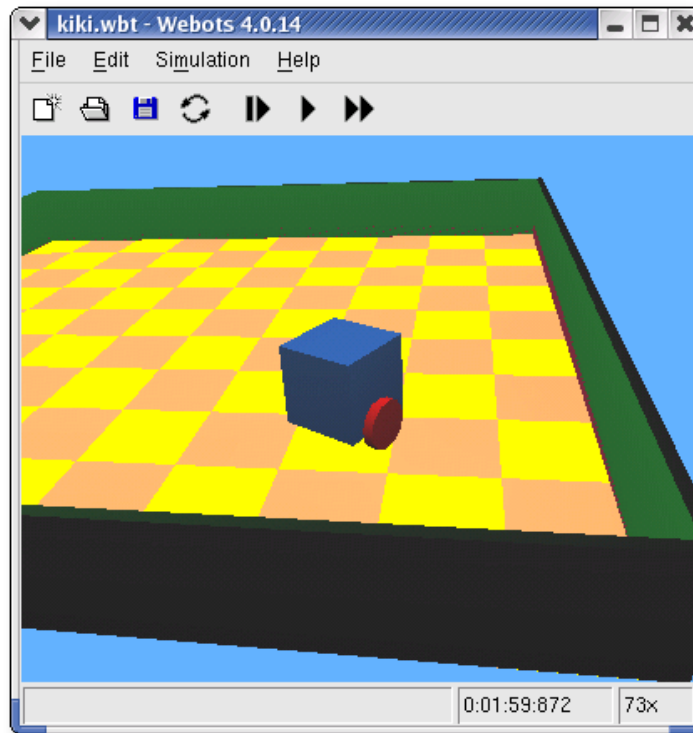


Figure 3.8: Body and wheels of the *kiki* robot

```
0.15    0  0 ]
```

8. To model the second IR sensor, select the `DistanceSensor` node and **insert after** a new `DistanceSensor` node. Type "ir1" as a name. Set its translation to `[0.02 0.08 -0.042]` and its rotation to `[0 1 0 1.57]`. In the children, **insert after** `USE INFRARED`. In the `lookupTable` field, type the same values as shown above.

The robot and its two sensors are shown in figure 3.10 and figure 3.11.

Note: A texture can only be mapped on an `IndexedFaceSet` shape. The `texCoord` and `texCoordIndex` entries must be filled. The image used as a texture must be a `.png` or a `.jpg` file, and its size must be $(2^n) * (2^n)$ pixels (for example 8x8, 16x16, 32x32, 64x64, 128x128 or 256x256 pixels). Transparent images are not allowed in Webots. Moreover, PNG images should use either the 24 or 32 bit per pixel mode (lower `bpp` or gray levels are not supported). Beware of the maximum size of texture images depending on the 3D graphics board you have: some old 3D graphics boards are limited to 256x256 texture images while more powerful ones will accept 2048x2048 texture images.

To paste a texture on the face of the robot:

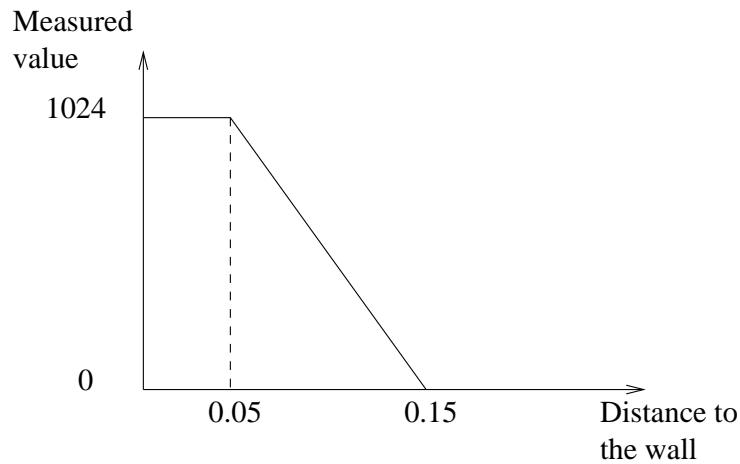


Figure 3.9: Distance measurements of the *kiki* sensors.

1. Select the last `DistanceSensor` node and **Insert after** a `Shape` node.
2. Create an `Appearance` node in the `appearance` field. Create an `ImageTexture` node in the `texture` field of this node, with the following URL: `"kiki/kiki.png"`. This refers to an image file lying in the `worlds` directory.
3. In the `geometry` field, create an `IndexedFaceSet` node, with a `Coordinate` node in the `coord` field. Type the coordinates of the points in the `point` field:

```
[ 0.015  0.05  -0.041,
  0.015  0.03  -0.041,
 -0.015  0.03  -0.041,
 -0.015  0.05  -0.041 ]
```

and **Insert after** in the `coordIndex` field the following values: `0, 1, 2, 3, -1`. The optional `-1` value is there to mark the end of the face. It is useful when defining several faces for the same `IndexedFaceSet` node.

4. In the `texCoord` field, create a `TextureCoordinate` node. In the `point` field, enter the coordinates of the texture:

```
[ 0  0
  1  0
  1  1
  0  1 ]
```

and in the `texCoordIndex` field, type `3, 0, 1, 2`. This is the standard VRML97 way to explain how the texture should be mapped to the object.

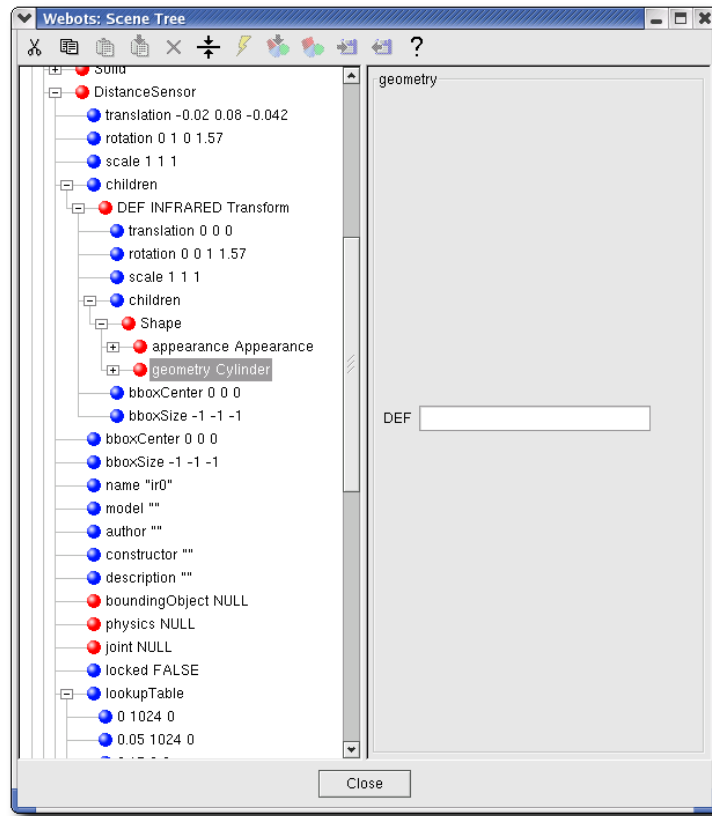


Figure 3.10: The DistanceSensor nodes of the *kiki* robot

5. The texture values are shown in figure 3.12.

To finish with the `DifferentialWheels` node, you must fill in a few more fields:

1. In the `controller` field, type the name "simple". It is used to determine which controller program controls the robot.
2. The `boundingObject` field can contain a `Transform` node with a `Box`, as a box as a bounding object for collision detection is sufficient to bound the *kiki* robot. Create a `Transform` node in the `boundingObject` field, with the `translation` set to `[0 0.05 -0.002]` and a `Box` node in its children. Set the dimension of the `Box` to `[0.1 0.1 0.084]`.
3. In the `axleLength` field, enter the length of the axle between the two wheels: 0.09 (according to figure 3.5).
4. In the `wheelRadius` field, enter the radius of the wheels: 0.025.
5. Values for other fields are shown in figure 3.13 and the finished robot in its world is shown in figure 3.14.

The `kiki.wbt` is included in the Webots distribution, in the `worlds` directory.

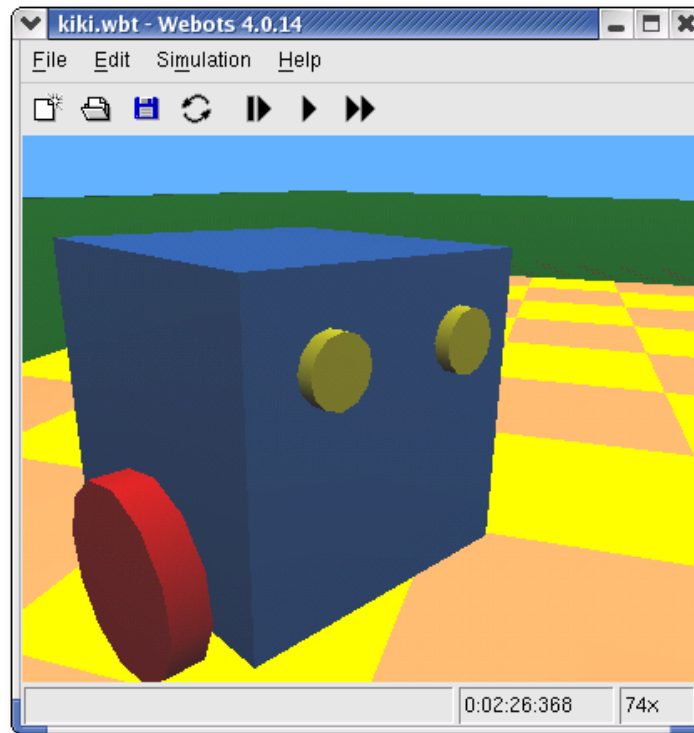


Figure 3.11: The *kiki* robot and its sensors

3.1.4 A simple controller

This first controller is very simple and thus named `simple`. The controller program simply reads the sensor values and sets the two motors speeds, in such a way that *kiki* avoids the obstacles.

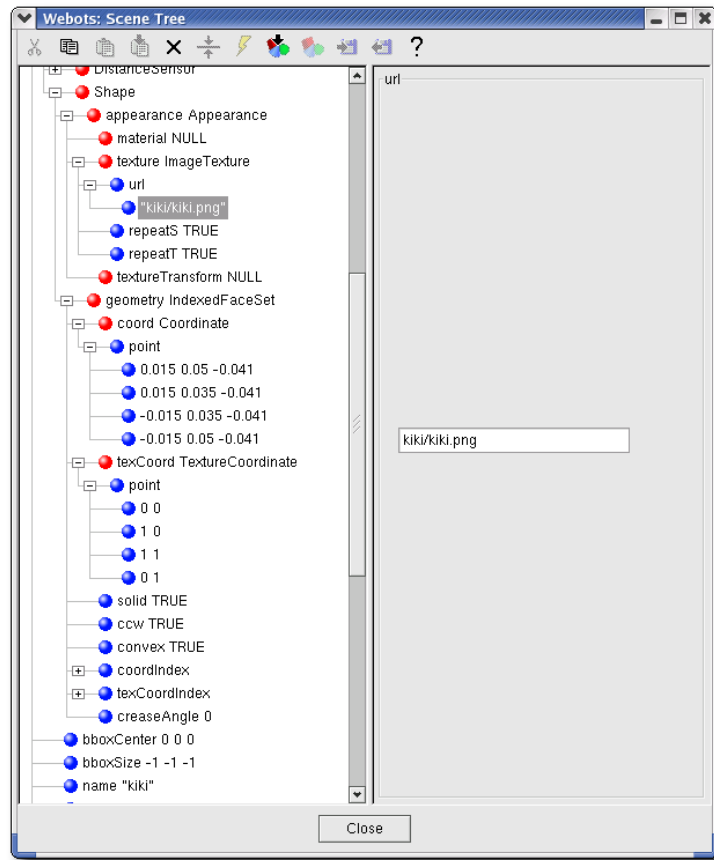
Below is the source code for the `simple.c` controller:

```
#include <device/robot.h>
#include <device/differential_wheels.h>
#include <device/distance_sensor.h>

#define SPEED 100

static DeviceTag ir0,ir1;

static void reset(void) {
    ir0 = robot_get_device("ir0");
    ir1 = robot_get_device("ir1");
    // printf("ir0=%d ir1=%d\n",ir0,ir1);
    distance_sensor_enable(ir0,64);
    distance_sensor_enable(ir1,64);
}
```

Figure 3.12: Defining the texture of the *kiki* robot

```

}

static void run(int ms) {
    short left_speed, right_speed;
    unsigned short ir0_value, ir1_value;

    ir0_value = distance_sensor_get_value(ir0);
    ir1_value = distance_sensor_get_value(ir1);
    if (ir1_value > 200) {
        left_speed = -20;
        right_speed = 20;
    } else if (ir0_value > 200) {
        left_speed = 20;
        right_speed = -20;
    } else {
        left_speed = SPEED;
        right_speed = SPEED;
    }
}

```

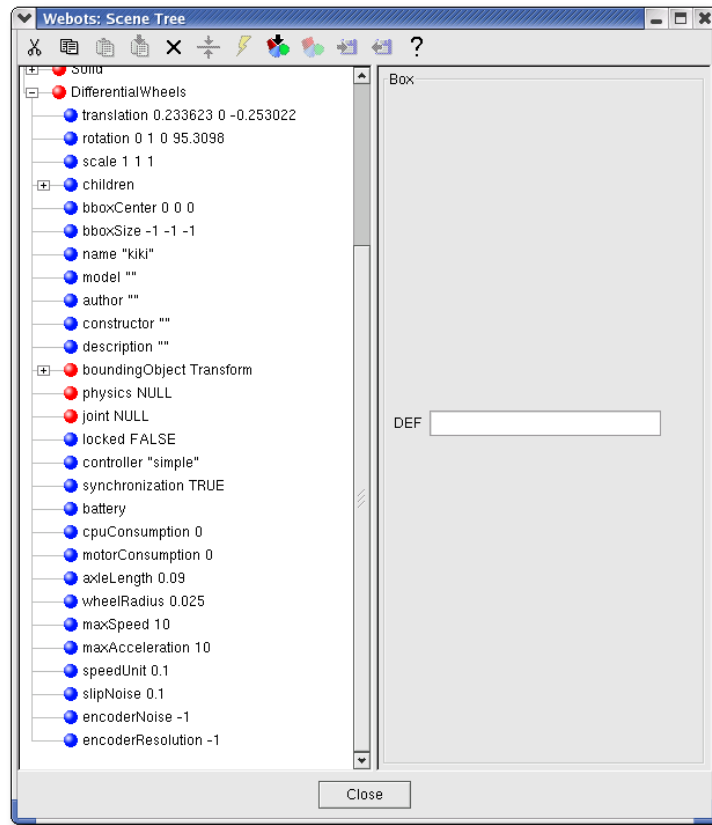


Figure 3.13: The other fields of the DifferentialWheels node

```

}
/* Set the motor speeds */
differential_wheels_set_speed(left_speed, right_speed);
return 64; /* next call after 64 milliseconds */
}

int main() {
  robot_live(reset);
  robot_run(run); /* this function never returns */
  return 0;
}

```

This controller lies in the `simple` directory of the Webots controllers directory.

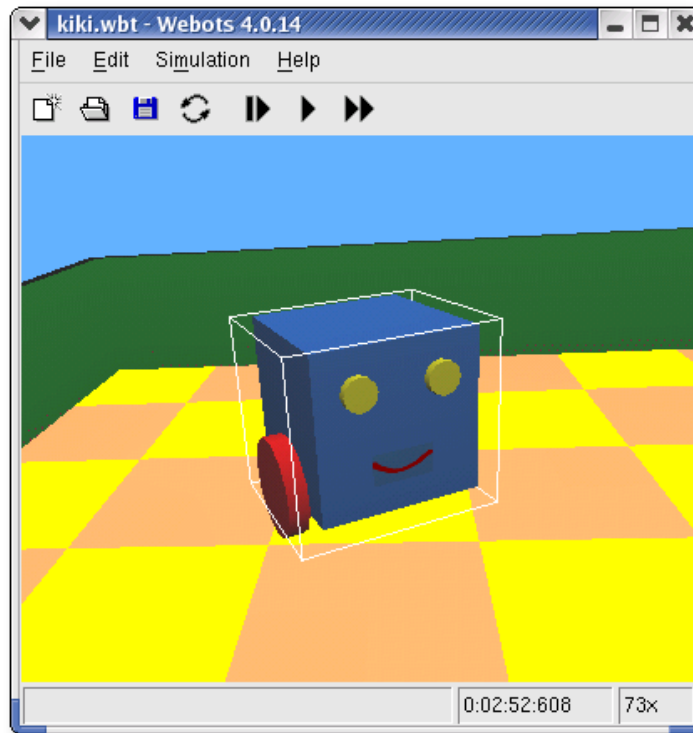


Figure 3.14: The *kiki* robot in its world

3.2 Adding a camera to the *kiki* robot

This section can be considered as an exercise to check if you understood the principles for adding devices to a robot. If you want to skip this section because you feel comfortable with Webots and you are not interested in cameras, you may jump directly to the next section which addresses physics and does not require that the *kiki* be equipped with a camera.

The camera to be modeled is a color 2D camera, with an image 80 pixels wide and 60 pixels high, and a field of view of 60 degrees (1.047 radians).

We can model the camera shape as a cylinder, on the top of the *kiki* robot at the front. The dimensions of the cylinder are 0.01 for the radius and 0.03 for the height. See figure 3.15.

Try modeling this camera. The `kiki_camera.wbt` file is included in the Webots distribution, in the `worlds` directory, in case you need any help.

A controller program for this robot, named `camera` is also included in the Webots distribution, in the `controllers` directory. This camera program actually do not perform image processing since it is just a demonstration program, but you could easily extend it to perform actual image processing. It would be useful then to add extra objects in the world, so that the robot could for example learn to recognize them and move towards or away from them depending if the object is categorized as good or bad.

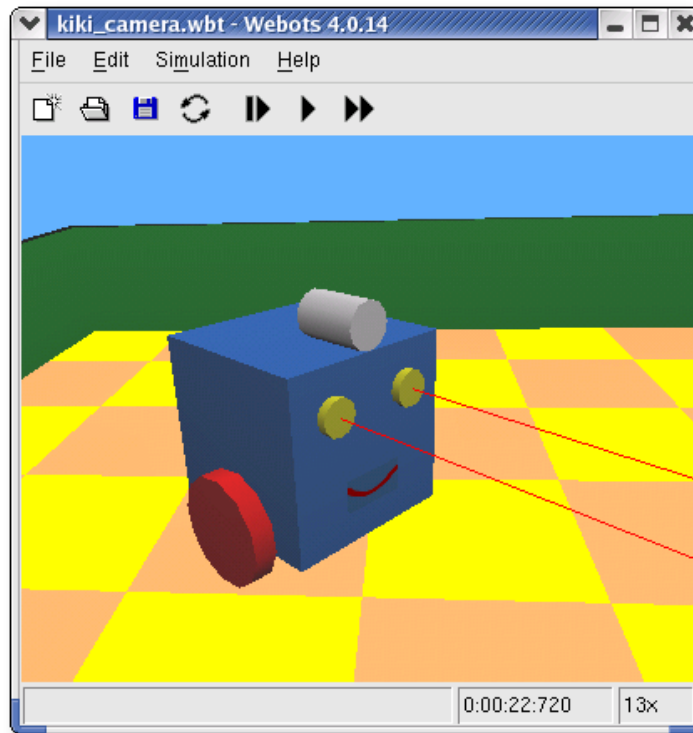


Figure 3.15: The *kiki* robot with a camera

3.3 Adding physics to the *kiki* simulation

3.3.1 Overview

The current model we defined for the *kiki* robot doesn't include any physics modelling, as we didn't specified any mass for example. Instead it is a simple kinematic model which can be used nonetheless for many mobile robotics simulation experiments where inertia and friction can be neglected. For example, it is well suited to simulate light desktop robots like Khepera or Hemisson. Finally, simulations run faster without physics.

However, as soon as things get more complex, you will need to introduce some physics in your model. For example, if your robot is heavy, you cannot afford to neglect inertia effects on its trajectory. If you want to add moveable objects, like boxes or a ball, physics simulation turn out to be necessary. Finally, if you want to model a robot architecture different from the plain differential wheels model, like a omni-directional robot, a legged robot, a swimming robot or a flying robot, then you need to setup many physics parameter.

This section introduces a simple physics simulation to the *kiki* world allowing the robot to play with a ball. More complex physics simulations can be implemented with Webots, involving different locomotion schemes based on the `CustomRobot` and `Servo` nodes, allowing to build

complex wheeled and legged robots. Other possibilities include flying and swimming robots where hydrodynamics models are needed. These features won't be addressed in this tutorial. Instead, it is recommended that you study the existing examples of legged and flying robots included within the Webots distribution, and refer to the documentation of the `CustomRobot` and `Servo` nodes. Do not hesitate to contact us if you need some support implementing complex physics in your simulation.

3.3.2 Preparing the floor for a physics simulation

Select the floor node which should be the first `Transform` node in the scene tree just after the `PointLight` nodes. Turn that `Transform` into a `Solid` node using the **Transform** button (representing a lightning).

Now, it is possible to define a `boundingObject` for the floor. Create an `IndexedFaceSet` node as bounding object. In this node, create a `Coordinate` node for the `coord` field. This node should define the following point list: `[1 0 1] [1 0 0] [0 0 0] [0 0 1]`. The `coordIndex` should contain the 0, 1, 2 and 3 values. This defines a square corresponding to the `ElevationGrid` of the floor. The bounding object we just defined will prevent the robot from falling down through the floor as a result of the gravity.

3.3.3 Adding physics to the *kiki* robot

The *kiki* robot already has a bounding object defined. However, since it will be moving, it also needs physics parameters that will be defined in its `physics` field as a `Physics` node. Create such a node and set its `density` to 100. The density is expressed in kilogram per cubic meter. Leave the `mass` to -1, as it is ignored when the density is specified. If ever you wanted to use the mass instead of the density, set the density to -1 and set the mass to a positive value. The mass is expressed in kilograms. However, for the rest of this tutorial, it is recommended to follow the guide and set the density as requested, leaving the mass to -1.

Now the wheels of the robot also need some physics properties to define the friction with the floor. But first they need a bounding object. Set the defined `WHEEL` node as the `boundingObject` for each wheel `Solid`. Then, add a `Physics` to the first wheel, write `WHEEL_PHYSICS` as a `DEF` name. Set the `density` to -1, the `mass` to 0.01, the `coulombFriction` to 0.9 and the `forceDependantSlip` to 0.1. Use this `WHEEL_PHYSICS` definition to define the physics of the second wheel. Finally, add a `Joint` node to the `joint` field of each wheel. This means that each wheel is connected to the robot body through a joint.

We are now done! Save the world as `my_kiki_physics.wbt`, reload it using the `revert` button and run the simulation. You will observe that the robot is moving not very steadily (especially if you look at what the robot's camera sees). That's physics! Of course you can improve the stability of the movement by adjusting the bounding object of the robot, the speed of the wheels, the friction parameters, etc.

3.3.4 Adding a ball in the *kiki* world

Now let's offer a toy to our robot. Instead of creating a ball object from scratch, let's borrow it from another world where such an object already exists. Open the `soccer.wbt` world. Double-click on the soccer. This should open the scene tree window and select the BALL solid. Simply copy it from the **Copy** button and re-open your `kiki_physics.wbt` world. Open the scene tree window, select the last object of the scene tree and click on the **Paste after**. Can you see the soccer ball? Read **How do I move an object?** from the **Help** menu and place the ball in front of the robot. Save the world and run the simulation. The *kiki* robot should be able to kick the ball, making it roll and bounce on the walls.

3.4 Modelling an existing robot: `pioneer2.wbt`

We are now going to model and simulate a commercial robot from ActivMedia Robotics: Pioneer 2-DX™, as shown on the ActivMedia Web site: <http://www.activrobots.com>. First, you must model the robots environment. Then, you can model a Pioneer 2™ robot with 16 sonars and simulate it with a controller.

Please refer to the `worlds/pioneer2.wbt` and `controllers/pioneer2` files for the world and controller details.

3.4.1 Environment

The environment consists of:

- a chessboard: a `Solid` node with an `ElevationGrid` node.
- a wall around the chessboard: `Solid` node with an `Extrusion` node.
- a wall inside the world: a `Solid` node with an `Extrusion` node.

This environment is shown in figure 3.16.

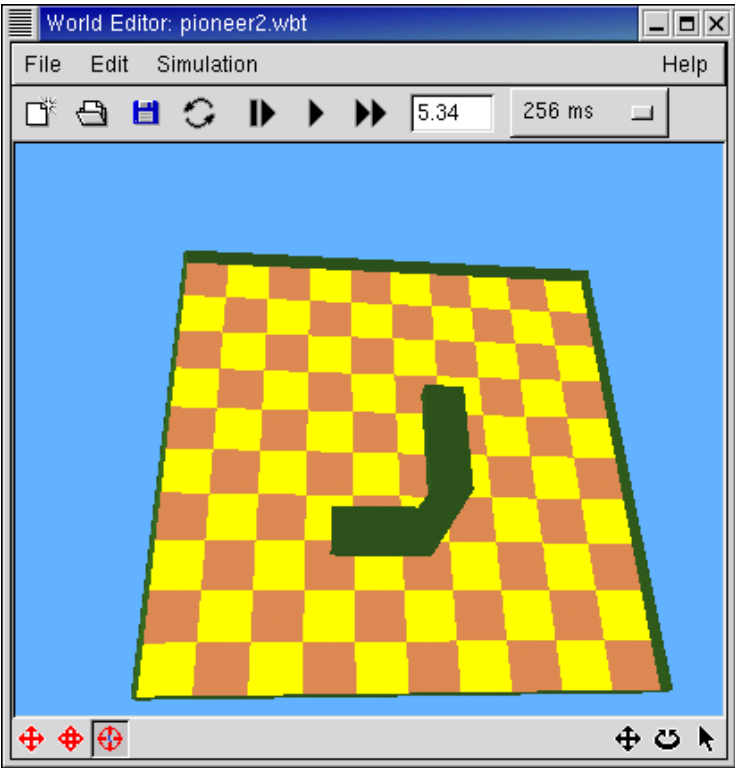


Figure 3.16: The walls of the Pioneer 2™ robot world

3.4.2 Robot with 16 sonars

The robot (a `DifferentialWheels` node) is made up of six main parts:

1. the body: an `Extrusion` node.
2. a top plate: an `Extrusion` node.
3. two wheels: two `Cylinder` nodes.
4. a rear wheel: a `Cylinder` node.
5. front an rear sensor supports: two `Extrusion` nodes.
6. sixteen sonars: sixteen `DistanceSensor` nodes.

The Pioneer 2 DXTM robot is depicted in figure 3.17.

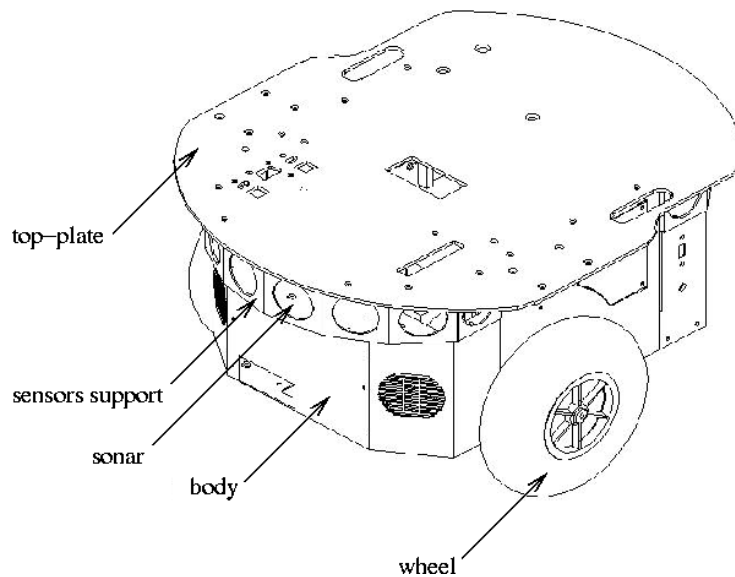


Figure 3.17: The Pioneer 2 DXTM robot

Open the tree editor and add a `DifferentialWheels` node. **Insert** in the children field:

1. for the body: a `Shape` node with a geometry `Extrusion`. See figure 3.18 for the coordinates of the `Extrusion`.
2. for the top plate: a `Shape` node with a geometry `Extrusion`. See figure 3.19 for the coordinates of the `Extrusion`.

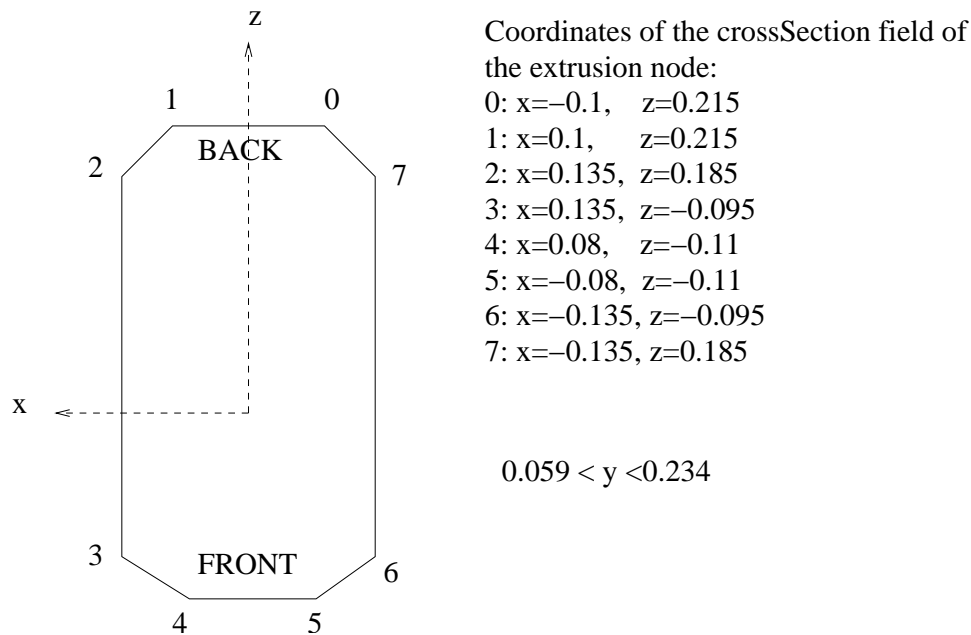


Figure 3.18: Body of the Pioneer 2™ robot

- for the two wheels: two `Solid` nodes. Each `Solid` node children contains a `Transform` node, which itself contains a `Shape` node with a geometry `Cylinder`. Each `Solid` node has a name: "left wheel" and "right wheel". See figure 3.20 for the wheels dimensions.
- for the rear wheel: a `Transform` node containing a `Shape` node with a geometry field set to `Cylinder`, as shown in figure 3.21
- for the sonar supports: two `Shape` nodes with a geometry `Extrusion`. See figure 3.22 for the `Extrusion` coordinates.
- for the 16 sonars: 16 `DistanceSensor` nodes. Each `DistanceSensor` node contains a `Transform` node. The `Transform` node has a `Shape` node containing a geometry `Cylinder`. See figure 3.23 and the text below for more explanation.

Modeling the sonars:

The principle is the same as for the *kiki* robot. The sonars are cylinders with a radius of 0.0175 and a height of 0.002. There are 16 sonars, 8 on the front of the robot and 8 on the rear of the robot (see figure 3.23). The angles between the sonars and the initial position of the `DEF SONAR` `Transform` are shown in figure 3.24. A `DEF SONAR` `Transform` contains a `Cylinder` node in a `Shape` node with a rotation around the z axis. This `DEF SONAR` `Transform` must be rotated and translated to become the sensors `FL1`, `RR4`, etc.

Each sonar is modeled as a `DistanceSensor` node, in which can be found a rotation around the y axis, a translation, and a `USE SONAR` `Transform`, with a name (`FL1`, `RR4`, ...) to be used

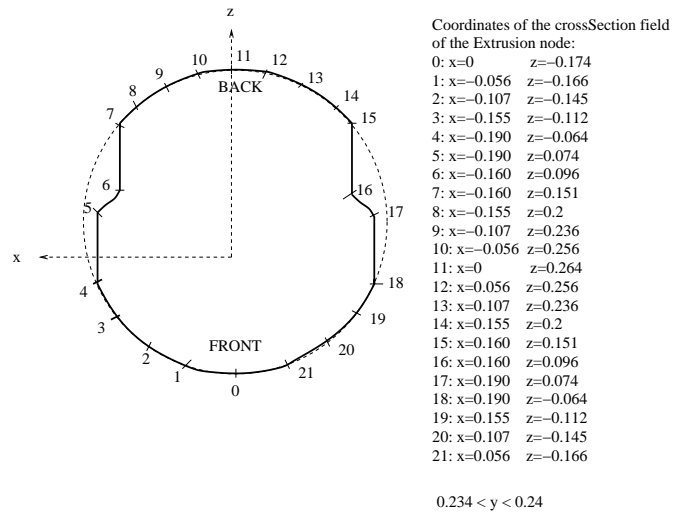


Figure 3.19: Top plate of the Pioneer 2™ robot

by the controller.

To finish modeling the Pioneer 2™ robot, you will have to fill in the remaining fields of the DifferentialWheels node as shown in figure 3.25.

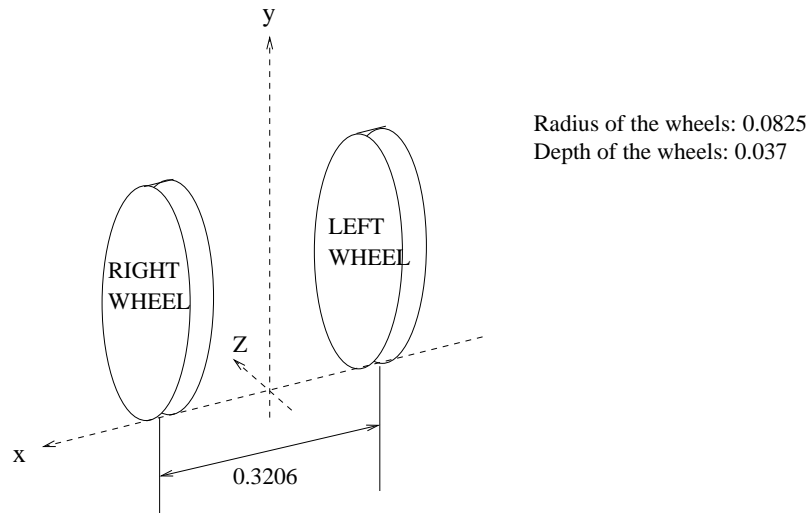


Figure 3.20: Wheels of the Pioneer 2TM robot

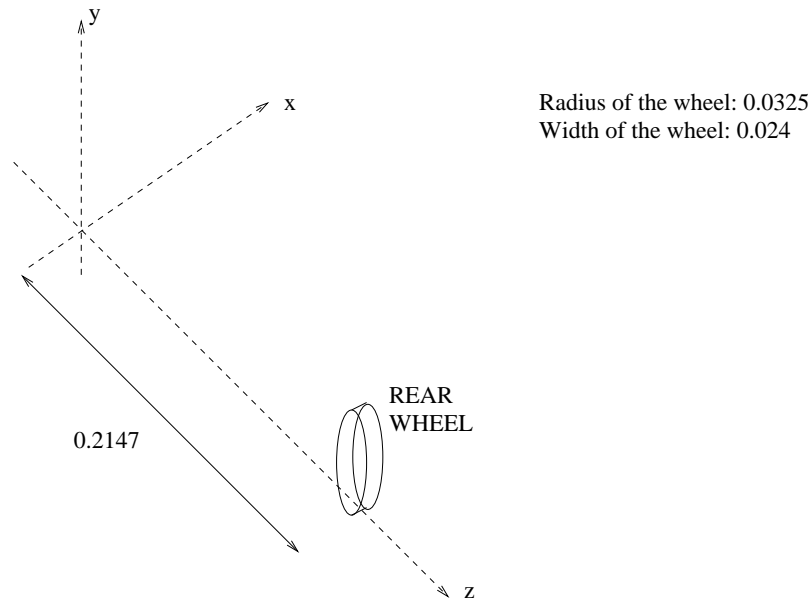


Figure 3.21: Rear wheel of the Pioneer 2TM robot

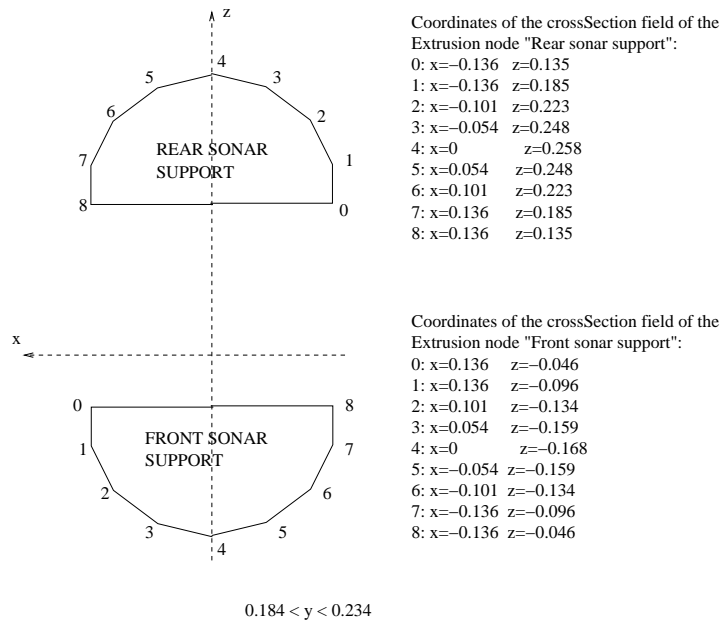


Figure 3.22: Sonar supports of the Pioneer 2™ robot

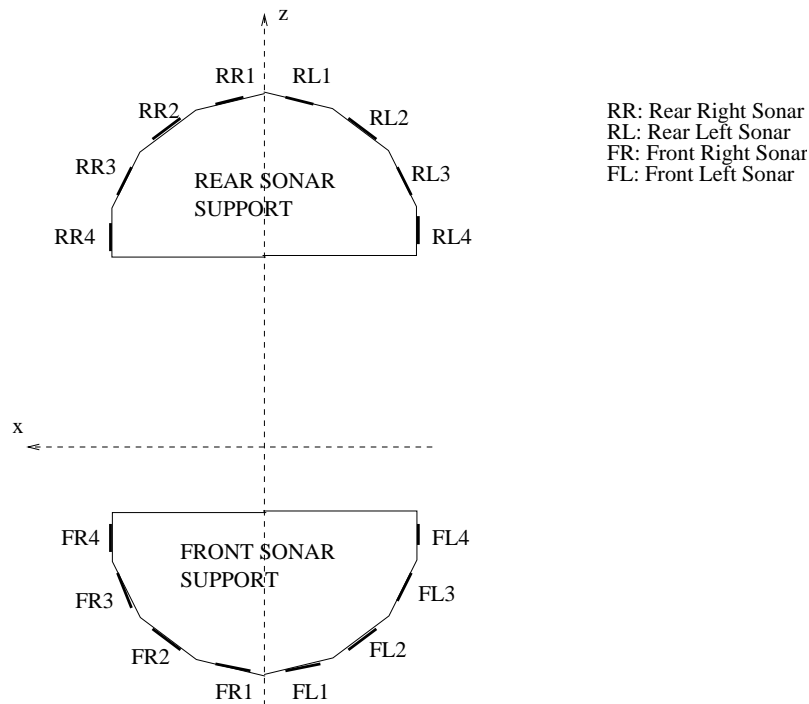


Figure 3.23: Sonars location on the Pioneer 2™ robot

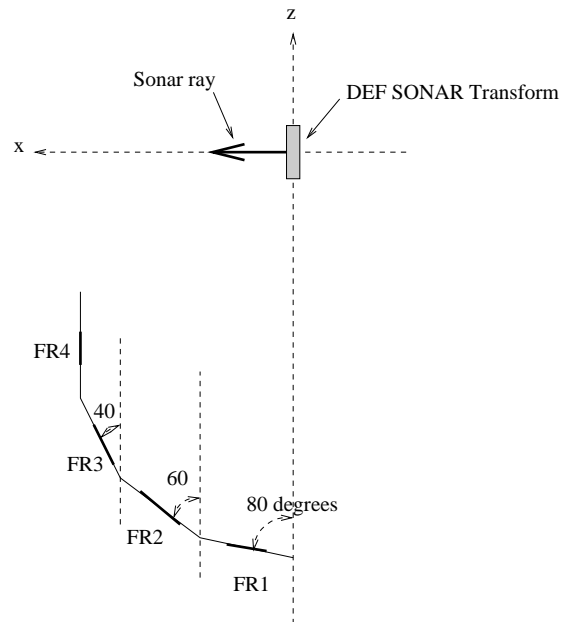


Figure 3.24: Angles between the Pioneer 2™ sonar sensors

Sonar name	translation	rotation
FL1	-0.027 0.209 -0.164	0 1 0 1.745
FL2	-0.077 0.209 -0.147	0 1 0 2.094
FL3	-0.118 0.209 -0.111	0 1 0 2.443
FL4	-0.136 0.209 -0.071	0 1 0 3.14
FR1	0.027 0.209 -0.164	0 1 0 1.396
FR2	0.077 0.209 -0.147	0 1 0 1.047
FR3	0.118 0.209 -0.116	0 1 0 0.698
FR4	0.136 0.209 -0.071	0 1 0 0
RL1	-0.027 0.209 0.253	0 1 0 -1.745
RL2	-0.077 0.209 0.236	0 1 0 -2.094
RL3	-0.118 0.209 0.205	0 1 0 -2.443
RL4	-0.136 0.209 0.160	0 1 0 -3.14
RR1	0.027 0.209 0.253	0 1 0 -1.396
RR2	0.077 0.209 0.236	0 1 0 -1.047
RR3	0.118 0.209 0.205	0 1 0 -0.698
RR4	0.136 0.209 0.160	0 1 0 0

Table 3.1: Translation and rotation of the Pioneer 2™ DEF SONAR Transforms

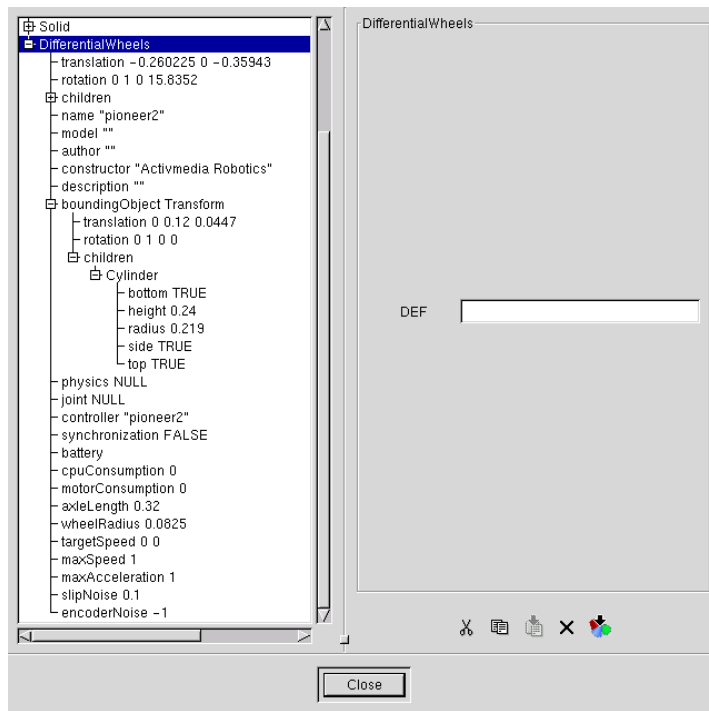


Figure 3.25: Some fields of the Pioneer 2TMDifferentialWheels node

3.4.3 Controller

The controller of the Pioneer 2TM robot is fairly complex. It implements a Braitenberg controller to avoid obstacles using its sensors. An activation matrix was determined by trial and error to compute the motor commands from the sensor measurements. However, since the structure of the Pioneer 2TM is not circular some tricks are used, such as making the robot go backward in order to rotate safely when avoiding obstacles. The source code of this controller is a good programming example. The name of this controller is `pioneer2`.

3.5 Transfer to your own robot

Mobile robot simulation is relevant because it is possible to transfer the results onto real mobile robots. Webots was designed with this transfer capability in mind. The simulation is as realistic as possible and the programming interface can be ported or interfaced to existing real robots. Webots already include transfer systems for a number of existing real robots including KheperaTM, HemissonTM, LEGO MindstormsTM, AiboTM, etc. This section explains how to develop your own transfer system to your very own mobile robot.

Since the simulation is always a more or less accurate approximation of the physics of the real robot, some tuning is always necessary when developing a transfer mechanism for an existing real robot. This tuning will affect the simulated model so that it better matches the behavior of the real robot.

3.5.1 Remote control

Overview

The easiest way to transfer your control program to a real robot is often to develop a remote control system. In this case, your control program runs on the computer, but instead of sending commands to and reading sensor data from the simulated robot, it sends commands to and reads sensor data from the real robot. Developing such a remote control system can be achieved in a very simple way by writing your own implementation of the Webots API functions as a small library. For example, you will probably have to implement the `differential_wheels_set_speed` function as a function that sends a specific command to the real robot with the wheel speeds as an argument. This command can be sent to the real robot via the serial port of the PC or whatever PC robot interface you have. You will probably need to make some unit conversion since your robot may not use the same speed unit as the one used in Webots. The same applies for reading sensor values from the real robot.

Developing a custom library

Once you have created a number of C functions implementing the Webots functions you need to redirect outputs and inputs to the real robot. You will then be able to reuse your Webots controller used for the simulation without changing a line of code, and even without recompiling it to an object file: Instead of linking this object file with the Webots `Controller` dynamic library, you will link it with your own C functions. For your convenience, you may want to create a static or dynamic library containing your own robot interface.

Special functions

The `robot_live` function can be used to perform some initialization, like setting up the connection with the real robot.

The `robot_get_device` function should return arbitrary integer values specific to each device of your real robot. These values should be used by device specific functions. For example, the `distance_sensor_get_value` function is able to recognize the specified device and return the correct value.

The `robot_run` function should call repeatedly the function passed as an argument. The first call should be performed with 0 as an argument. It should then take care of the return value of the run function and respect the requested delay before calling again this function. The parameter passed to the run function should describe the actual delay (see reference description about the `robot_run` function for more details about it).

Running your real robot

Once linked with your own library, your controller can be launched as a stand alone application to control your real robot. It might be useful to include in your library or in your Webots controller some graphical representation to display sensor values, motor commands or a stop button. Such a remote control system can be implemented in C as explained here, however, it can also be implemented in Java using the same principle by replacing the `Controller.jar` Webots file by your own robot specific `Controller.jar` file and using this one to drive the real robot.

3.5.2 Cross-compilation

Overview

Developing a cross-compilation system will allow you to recompile your Webots controller for the embedded processor of your own real robot. Hence the source code you wrote for the Webots simulation will be executed on the real robot itself and there is no need to have a permanent PC connection with the robot as with the remote control system. This is only possible if the processor

on your robot can be programmed in C, C++ or Java. It is not possible for a processor that can be programmed only in assembler or another specific language. Webots includes the source code of such a cross-compilation system for the Hemisson robot. This sample is located in the `Hemios` directory of the `hemisson controller`.

Developing a custom library

Unlike the remote control system, the cross-compilation system requires that the source code of your Webots controller be recompiled using the cross-compilation tools specific to your very own robot. You will also need to rewrite the Webots include files to be specific to your very own robot. In simple cases, you can simply rewrite the Webots include files you need, as in the `hemisson` example. In more complex cases, you will also need to write some C source files to be used as a replacement of the `Webots Controller` library, but running on the real robot. You should then recompile your Webots controller with your robot cross-compilation system and link it with your robot library. The resulting file should be uploaded onto the real robot for local execution.

Examples

Webots support cross-compilation for the several existing commercial robots. For the Hemisson™ robot, this system is as simple a few include files replacing the Webots API include files. For the Khepera™ robot, a specific C library is used additionally to specific include files. For the LEGO Mindstorms™ robot, a Java library is used and the resulting binary controller is executed on the real robot using the LeJOS Java virtual machine.

3.5.3 Interpreted language

In some cases, it may be better to implement an interpreted language system. This is useful if your real robot already uses an interpreted language, like Basic or a graph based control language. In such a case, the transfer is very easy since you will just transfer the code of your program that will be interpreted on the real robot. The most difficult part may be to develop a language interpreter in C or Java to be used by your Webots controller for controlling the simulated robot. Such an interpreted language system was developed for the Hemisson™ robot with the BotStudio™ system.

3.6 Adding custom ODE physics

3.6.1 Introduction

This section describes the capability to add custom physics simulation to your Webots simulations. This is especially useful if you want to model complex forces and torques, such as hydrodynamical forces or a random wind. It is also possible to gather various information (like the position, orientation, linear or angular velocity, etc. of every solid in the world or the global parameters of the physical simulation), to decide which force or torque should be applied. This way, it is possible to apply hydrodynamic forces only when a robot enters a special part of the world which is supposed to contain water. You may also access internal parameters of the physics engine for a better tuning of your physics simulation. Moreover, you can also implement your own collision detection system to better control contact joints and define for example non-uniform friction parameters on some surfaces.

Adding a custom physics is achieved by creating a custom shared library which is loaded by Webots at run-time and which contains function calls to the ODE physics library. This system currently runs on Linux, Windows and Mac OS X operating systems.

3.6.2 Files

The `WorldInfo` node of the simulated world has a field called `physics` which defines the name of the shared library to be used for the custom physics simulation in this world. This name has no extension such as `.so` (under Linux), `.DLL` (under Windows) or `.dylib` (under Mac OS X), but refers to a shared library stored in a subdirectory of the Webots user `physics` directory (at the same level as the `controllers` and the `worlds` directories). For example:

```
WorldInfo {
  physics "sample"
}
```

refers to the `sample.so` shared library under Linux, to the `sample.dll` shared library under Windows or to the `sample.dylib` shared library under Mac OS X. This shared library should be stored in the `sample` subdirectory of the Webots user `physics` directory.

Since the shared library for physics is referred to by the `WorldInfo` node of a world, you can develop different physics shared libraries for different worlds.

3.6.3 Implementation

Callback functions

Your shared library may contain four functions that will be called directly by Webots during the simulation of the world. You may implement all of these functions, or only a few of them. If the

functions are not implemented, they won't be called.

- `void webots_physics_init(dWorldID, dSpaceID, dJointGroupID);` This function is called upon initialization of the world. It provides your shared library with ODE variables used by the simulation, such as a pointer to the world (`dWorldID`), a pointer to the geometry space (`dSpaceID`) and a pointer to the contact joint group used by the simulation (`dJointGroupID`). All these parameters should be stored in global variables of your shared library for further use. Moreover, this function is a good place to call the `dWebotsGetGeomFromDEF` function (see below for details) to get pointers to the objects on which you want to control the physics. Before this function is called, the current directory is set to the directory in which your physics shared library lies. This is useful for reading config files or writing log files in this directory.
- `void webots_physics_step();` This function is called before every physics simulation step (call to the ODE `dWorldStep()` function). It has no parameter. It can be used to add force and / or torques to solids. It can also be used to test the position and orientation of solids (and possibly apply different forces according the position and orientation).
- `int webots_physics_collide(dGeomID, dGeomID);` This function is called whenever a collision occurs between two objects. It may be called several times for a single simulation step with different parameters corresponding to different objects. You should test whether the two colliding objects passed as arguments correspond to the objects you want to control. Then, you should create the contact joints, using the ODE `dCollide` and `dJointCreateContact` functions. Finally, you should add this contact joint to the joint group passed as an argument of the `webots_physics_init` function using the ODE `dJointAttach` function. Finally, you should return 1 if the collision has been handled by your function or 0 if you wish that Webots handle this collision using its default collision system.
- `void webots_physics_cleanup();` This function is the counterpart function of the `webots_physics_init` function. It is called when the world is destroyed and can be used to perform some cleanup, like releasing resources and so on.
- `void webots_physics_draw();` This function is a utility function intended to display additional 3D objects in the main 3D window. This is useful to display for example some forces as lines with arrows or to add some objects in the world. It is called immediately after the world is displayed. This function should contain OpenGL calls `glEnable`, `glDisable`, `glColor4f`, `glBegin`, `glVertex3f`, `glEnd`, etc. The OpenGL state should be restored to the default value at the end of this function to avoid subsequent rendering problems in Webots.

dWebotsGetGeomFromDEF

As mentioned in the description of the `webots_physics_init` function, a special function called `dWebotsGetGeomFromDEF` allows you to get a pointer (actually an ODE `dGeomID`) to a

`Solid` node of the world defined by its `DEF` name. The prototype for this function is:

```
dGeomID dWebotsGetGeomFromDEF(const char *DEF);
```

where `DEF` is the `DEF` name of the requested `Solid` node. From this `dGeomID` pointer, ODE allows you to obtain the corresponding `dBodyID` pointer using the ODE `dGeomGetBody` function.

dWebotsSend and dWebotsReceive

It is often useful to communicate some information between your custom physics library and robot (or supervisor) controllers. This is especially useful if your custom physics library implements some sensors (like accelerometers, force feedback sensors, etc.) and needs to send the sensor measurement to the robot controller. It is also useful if your custom physics library implements some actuators (like a linear servo or an Akermann drive model) and needs to receive motor commands from a robot controller.

The physics library API provides the `dWebotsSend` function to send messages to robots controller and the `dWebotsReceive` function to receive messages from robots controllers. In order to receive messages from the physics library, a robot has to contain a `Receiver` node set on an appropriate channel (see reference manual) and with a `baudRate` set to -1 (for infinite communication speed). Messages are sent from the physics library using `dWebotsSend` function and received through the receiver API as if they were sent by an `Emitter` node with an infinite range and baud rate. Similarly, in order to send messages to the physics library, a robot has to contain an `Emitter` node set on channel 0 (as the physics library only receive data sent on this channel). The `range` and `baudRate` fields of the `Emitter` node should be set to -1 (to be considered as infinite). Messages are sent to the physics library using the standard emitter API functions. They are received by the physics library through the `dWebotsReceive` function.

```
void dWebotsSend(int channel,void *buffer,int size);
void *dWebotsReceive(int *size);
```

The `dWebotsSend` function sends `size` bytes of data contained in `buffer` over the specified communication channel.

The `dWebotsReceive` function receives any data emitted on channel 0. If no data was emitted, it returns `NULL`, otherwise it returns a pointer to a buffer containing the received data. If `size` is non-`NULL`, it is set to the number of bytes of data available in the returned buffer. This buffer is currently limited to 1024 bytes.

3.6.4 Compiling the shared library

Your shared library can be compiled under Windows and Linux with GNU `make` and `gcc` using the provided `Makefile`. You can also use Visual C++ under Windows to compile it. Under

Windows, the shared library should be dynamically linked to the ODE library. The Webots `lib` directory contains the gcc (`libode.a`) and Visual C++ (`ode.lib`) import libraries. Under Linux, you don't need to link the shared library with anything.

3.6.5 Example

An example of custom physics shared library is provided within the `flying_robot.wbt` world which uses the `sample` physics shared library. You can read the source code of this library in the `sample` subdirectory of the Webots `physics` directory. In this example, the custom physics library is used to add some wind and to define a non-uniform friction between a cube robot and the floor.

Chapter 4

Robot and Supervisor Controllers

4.1 Overview

A robot controller is a program usually written in C, C++ or Java used to control one robot. A supervisor controller is a program usually written in C or C++ used to control a world and its robots.

4.2 Setting Up a Development Environment

4.2.1 Under Windows

Using MinGW with Webots built-in source code editor

MinGW is a free development environment based on the `gcc` open source C and C++ compiler. It includes the `make` utility used to compile the Webots controllers from the provided `Makefile` files. MinGW is included in the `devel` subdirectory of the `windows` directory on the Webots CD-ROM. You should install it in order to be able to compile Webots controllers from the `compile` button of the source code editor.

After installing MinGW, please check that the path to the MinGW `bin` directory (where `gcc` lies) is included in the `PATH` environment variable of your system. This path is usually something like `C:\mingw\bin`.

Also, if the **compile** button fails to compile your Webots controller, please check that no `sh.exe` file is reachable from your `PATH` environment variable. If this is the case, remove the path to this file from your `PATH` environment variable, or rename this `sh.exe` file to a different name.

Using MinGW with your own custom environment

MinGW comes optionally with a companion utility called MSYS which is a UNIX-like terminal that can be used to invoke the MinGW commands. MinGW should be installed prior to MSYS. In addition to MinGW and MSYS, you will probably need a text editor to write your controller programs. We recommend using SciTe, which is a simple, lightweight source code editor. SciTe is also provided in the `devel` subdirectory of the `windows` directory on the Webots CD-ROM. Alternatively to SciTe, you may want to use Dev-C++, which is a Visual C++ like development environment relying on `gcc`. Dev-C++ is also provided in the `devel` subdirectory of the `windows` directory on the Webots CD-ROM. A sample Dev-C++ project called `braiten.dev` is provided in the `braiten` controller directory of Webots.

Using Visual C++

Visual C++ is an integrated development environment for C and C++ provided by Microsoft Corp. It includes a C and C++ compiler and a source code editor. A number of Visual C++ project examples are provided in the `controllers/braiten`, `controllers/khepera` and `controllers/tcpip` controller directories. Typically, a new Visual C++ project for Webots should define a correct include path to the Webots `include` directory and should link the executable file with the `Controller.lib` file included in the Webots `lib` directory. Take care to produce an executable file in the specific controller directory and not in a `Debug` or `Release` subdirectory as produced by default by Visual C++. For example, the `khepera.exe` program should be created in the `khepera` directory of your `controllers` directory. Please note that the resulting executable files cannot be executed from Visual C++ as they should be launched by Webots and referenced in the world file used by Webots.

Here is the complete procedure to set up a new Webots controller project under Visual C++ 6.0:

1. Create a `my_controller` directory in your local `webots` directory. Launch Visual C++ and go to the **File New...** menu item.
2. Create a "Win32 Console Application" project (or "Win32 Application" if you don't need a console for debugging). Set the **Project name:** to `my_controller` and set the **Location:** to your local `webots\controllers\my_controller` directory. Choose to create an empty project.
3. Go to the **Files New...** menu item to create a new **C++ Source File** named `my_controller.c` in your `my_controller` directory.
4. Go to the **Build Configurations...** menu item and **Remove** the `Win32 Debug` configuration. Close the **Configurations** window.
5. Go to the **Projects Settings** menu item and select the **C/C++** tab. Select the **Preprocessor** category and type `C:\Program Files\Webots\include` in the **Additional include**

directories entry. Then, go to the **Link** tab, **General** category and replace the **Output file name:** `Release/my_controller.exe` by `my_controller.exe`. Then, prepend `Controller.lib` in the list of **Object/library modules:**. Finally, in the **Input** category, type `C:\Program Files\Webots\lib` as an **Additional library path:**

6. Now, type your Webots controller source code in the `my_controller.c` file (you can take inspiration from the `simple.c` controller provided in the `controllers` directory of Webots, usually located in `C:\Program Files\Webots\controllers\simple`).
7. Now build your application from the **Build Build my_controller.exe** menu item (or F7 key). It should create a `my_controller.exe` file in your `my_controller` directory. However, this binary file cannot be launched individually or from Visual C++. It has to be launched by a Webots world referring to that file.

Using the Java Development Kit

The Java Development Kit (JDK) is provided for free by Sun Microsystems. A copy of this development environment is included in the `devel` subdirectory of the `windows` directory on the Webots CD-ROM. It will allow you to program your Webots robots using the Java programming language. The Java Development Kit doesn't include any text editor or integrated development environment. You may use a simple text editor and invoke the `javac` Java compilation command from a DOS window, or use an integrated development environment like Borland JBuilder or Sun's NetBeans. If you installed MinGW, you will be able to invoke the `make` from a terminal which will in turn invoke the `javac` command appropriately.

4.2.2 Under Linux

This is the most simple case. Usually, you don't have to do anything since most Linux distributions come with the `gcc` C/C++ compiler and the `make` utility. If these tools are not installed, you will have to install them. Please refer to your Linux distribution to install them. Of course, you will also need a text editor or possibly an integrated development environment. We recommend using `emacs` as a text editor as it is very common under Linux.

If you want to program your robots using the Java language, you will have to install the Java Development Kit (JDK) from Sun Microsystems. This software is available for free from Sun Microsystems. It is also included in the `devel` subdirectory of the `linux` directory on the Webots CD-ROM.

4.2.3 Under Mac OS X

Simply installing the Developer Tools provided with Mac OS X allows you to program your Webots robots in C, C++ and Java. The Apple Developer Tools for Mac OS X include the `gcc`

C and C++ compiler, the `make` build command and the `javac` Java compiler. You will probably use the Project Buidler application to write your source code and the Terminal application to run the `make` command which will in turn invoke either the `gcc` or `javac` compiler.

4.3 Setting Up a New Controller

In order to develop a new controller, you must first create a `controllers` directory in your user directory to contain all your robot and supervisor controller directories. Each robot or supervisor controller directory contains all the files necessary to develop and run a controller. In order to tell Webots where your controllers are, you must set up your user directory in the Webots preferences. Webots will first search for a `controllers` directory in your user directory, and if it doesn't find, it will then look in its own `controllers` directory. Now, in your newly created `controllers` directory, you must create a controller subdirectory, let's call it `simple`. Inside `simple`, several files must be created:

- a number of C source files, like `simple.c` which will contain your code.
- a `Makefile` which can be copied (or inspired) from the Webots `controllers` directories. Note that Windows users also have several other alternatives to the `Makefile`: They can use a Dev-C++ project or a Microsoft Visual C++ project.

You can compile your program by typing `make` in the directory of your controller.

As an introduction, it is recommended that you copy the `simple` controller directory from the Webots `controllers` to your own `controllers` directory and then try to compile it.

Under Windows, if you use `make` and would like that your controller program opens up a DOS console to display `printf` messages, add the following line in your `Makefile`:

```
DOS_CONSOLE=1
```

4.4 Webots Execution Scheme

4.4.1 From the controller's point of view

Each robot controller program is built in the same manner. An initialization with the function `robot_live` is necessary before starting the robot. A callback function is provided to the `robot_live` function in order to perform some initialization of your controller program. In this initialization function, you will be able to identify the devices of the robot (see section 4.5) and to enable them.

Then, you should use the `robot_run` to declare your `run` that will be called immediately after the initialization function and at each control step until the simulator decides to terminate the simulation. This `run` must return an integer value that determines the duration of one step of the control loop, that is the number of simulated milliseconds after which the `run` function will be called again. The `run` function is used to retrieve sensor information, process it, compute motor commands and send motor commands.

The `robot_step` function is now deprecated and should not be used any more in new controller programs. Instead, new controller programs should rely on the `robot_run` function.

4.4.2 From the point of view of Webots

Startup

For each robot, Webots looks in the user `controllers` directory for a controller file matching the name specified as the controller of the robot. If the specified controller is `simple`, Webots will first try to execute the file called `simple` (on Linux or Mac OS X) or `simple.exe` (on Windows) located in the `simple` subdirectory of the user `controllers` directory. If such a file doesn't exist, then, it will look for a file called `simple.class` in the same subdirectory and launch it as a Java controller. If doesn't exist, then it will try to look for a file called `simple.jar` in the same directory and launch the `simple` class from it. If this one doesn't exist, then Webots will fail launching the specified controller and will use the `void` instead.

In case of a Java controller, all the `jar` files located in the specified controller directory will be added to the Java `CLASSPATH`. The only exception to this rule is that if a `jar` file has the same name as a `class` file in the same directory, then, this `jar` file will be ignored. This means that if you have both a `simple.jar` file and a `simple.class` file in the same directory, then the `simple.jar` file will not be added to the `CLASSPATH`. However, other `jar` files (if existing) will be added to the `CLASSPATH`.

Simulation loop

Webots receives controller requests from possibly several robots controllers. Each request is divided into two parts: an actuator command part which takes place immediately, and a sensor measuring part which is scheduled to take place after a given number of milliseconds (as defined by the parameter of the step function). Each request is queued in the scheduler and the simulator advances the simulation time as soon as it receives new requests.

4.4.3 Synchronous versus Asynchronous controllers

Each robot (`DifferentialWheels` or `Supervisor`) may be synchronous or asynchronous. Webots waits for the requests of synchronous robots before it advances the simulation time; it

doesn't wait for asynchronous ones. Hence an asynchronous robot may be late (if the controller is computationally expensive, or runs on a remote computer with a slow network connection). In this case, the actuator command occurs later than expected. If the controller is very late, the sensor measurement may also occur later than expected. However, this delay can be verified by the robot controller by reading the return value of the `robot_step` function (see the Reference Manual for more details). In this way the controller can adapt its behavior and compensate.

Synchronous controllers are recommended for robust control, while asynchronous controllers are recommended for running robot competitions where computer resources are limited, or for networked simulations involving several robots dispatched over a computer network with an unpredictable delay (like the Internet).

4.5 Reading Sensor Information

To obtain sensor information, the sensor must be:

1. *identified*: this is performed by the `robot_get_device` function which returns a handler to the sensor from its name. This needs to be done only once in the reset callback function, which is provided as an argument to the `robot_live` function. The only exception to this rule concerns the root device of a robot (DifferentialWheels or CustomRobot node) which doesn't need to be identified, because it is the default device (it always exists and there is only one of such device in each robot).
2. *enabled*: this is performed by the appropriate `enable` function specific to each sensor (see `distance_sensor_enable` for example). It can be done once, before the endless loop, or several times inside the endless loop if you decide to disable and enable the sensors from time to time to save computation time.
3. *run*: this is performed by the `robot_step` function inside the endless loop.
4. *read*: finally, you can read the sensor value using a sensor specific function call, like `distance_sensor_get_value` inside the endless loop.

4.6 Controlling Actuators

Actuators are easier to handle than sensors. They don't need to be enabled. To control an actuator, it must be:

1. *identified*: this is performed by the `robot_get_device` function which returns a handler to the actuator from its name. This needs to be done only once in the reset callback function, which is provided as an argument to the `robot_live` function. As with sensors, the only exception to this rule concerns the root device of a robot.

2. *set*: this is performed by the appropriate `set` function specific to each actuator (an example of such a function is `differential_wheels_set_speed`). It is usually called in the endless loop with different computed values at each step.
3. *run*: this is done outside the `robot_run` function.

4.7 Going further with the Supervisor Controller

The supervisor can be seen as a super robot. It is able to do everything a robot can do, and more. This feature is especially useful for sending messages to and receiving messages from robots, using the `Receiver` and `Emitter` nodes. Additionally, it can do many more interesting things. A supervisor can move or rotate any object in the scene, including the `Viewpoint`, change the color of objects, and switch lights on and off. It can also track the coordinate of any object which can be very useful for recording the trajectory of a robot. As with any C program, a supervisor can write this data to a file. Finally, the supervisor can also take a snapshot of the current scene and save it as a `jpeg` or `PNG` image. This can be used to create a "webcam" showing the current simulation in real-time on the Web!

4.8 Interfacing Webots to third party software

4.8.1 Overview

If you don't want to develop your robot controllers using C, C++ or Java, it is possible to interface Webots to almost any third party software, such as `MatLabTM`, `LispTM`, `LabViewTM`, etc. Such an interface is implemented through a TCP/IP protocol that you can define by yourself. Webots comes with an example of interfacing a simulated Khepera robot through TCP/IP to any third party program able to read from and write to a TCP/IP connection. This example world is called `tcpip.wbt` and lies in the `worlds` directory of Webots. The simulated Khepera robot is controlled by the `tcpip` controller which lies in the `controllers` directory of Webots. This small C controller comes with full source code in `tcpip.c`, so that you can improve it to suit your needs. A client example is provided as a binary and C source code in `client.c`. Such a client should be used as a model to rewrite a similar client using the programming language of your third party software. This has already been implemented in Lisp and MatLab by some Webots customers.

4.8.2 Main advantages

There are several advantages of using such an interface. First, you can have several simulated robots in the same world using the several instances of the same `tcpip` controller, each one using

a different TCP/IP port, thus allowing your third party software to control several robots through several TCP/IP connections. To allow the `tcpip` process to open a different port depending on the controlled robot, you should give a different name to each robot and use the `robot_get_name` in the `tcpip` controller to retrieve this name and decide to open a port specific for each robot.

The second advantage is that you can also remote control a real Khepera robot from your third party software without writing a line of code. Simply switching to the remote control mode in the Khepera window will redirect the input/output to the real robot through the serial line.

The third advantage is that you can spread your controller programs over a network of computers. This is especially useful if the controller programs perform computer expensive algorithms such as genetic algorithms or other learning techniques.

Finally, it should be mentioned that it might be interesting to set the controlled robot in synchronous or asynchronous mode depending if you want the Webots simulator waits for commands from your controllers or not. In synchronous mode (set the `synchronization` field of your robots to `TRUE`), the simulator will wait for commands from your controllers. The controller step defined by the `robot_step` parameter the `tcpip` controller will be respected. In asynchronous mode (set the `synchronization` field of your robots to `FALSE`), the simulator will run as fast as possible, without waiting for commands from your controllers. In the latter case, it might be interesting to check the `runRealTime` field of the `WorldInfo` node in the scene tree window to have a real time simulation in which robots should behave like a real robots controlled through an asynchronous connection.

4.8.3 Limitations

The main drawback of this method is that if your robot has a camera device, the protocole should send the images to the controller over TCP/IP, which might be pretty network intensive. Hence it is recommended to have a high speed network, or use small resolution camera images, or compress the image data before sending it to the controller. This overhead is negligible if you use low resolution cameras such as the Khepera K213.

4.8.4 MatLab™ TCP/IP utility

The standard version of MatLab™ doesn't provide a plain TCP/IP interface. However, a free toolbox called TCP/UDP/IP Toolbox 2.0.5 developed by Mr. Peter Rydesäter is available. This toolbox can be found on the Webots CD-ROM (in the `common util` directory), as well as on the MatLab web site. It is known to run on Windows, Linux and other UNIX systems. It can be used so that your MatLab programs can connect to the `tcpip` Webots controllers to drive robots.

4.8.5 MatLab™ C interface

Another option with MatLab™ is to use the MatLab™ C interface to connect a Webots controller to the MatLab™ engine. Any data can be passed to and from MatLab™, including for example the image from a Webots camera, and MatLab™ commands can be invoked from within the Webots controller to use that data. The only tricky part is converting the data between native C and MatLab types - this requires using the `mxArray` functions. The MatLab™guide¹ explains this in details in the chapter entitled "Calling MATLAB from C and Fortran Programs".

4.9 Programming Webots controllers with URBI

4.9.1 Introduction to URBI

What is URBI ?

URBI is the *Universal Robotic Body Interface*, developed in the Laboratory of Electronics and Computer Engineering² of ENSTA National Institute of Advanced Technologies³ by J.-C. Bailie. You can find more information about URBI on its home page⁴.

URBI is a simple yet powerful scripted language meant to be used as a convenient interface to robots' bodies. URBI is designed to work over a client/server architecture, the robot being the server, and the actual controller being the client. However it is possible to make the client run on-board, or to run URBI scripts directly on the server-side, without a client.

URBI is not designed to write complex or computationnaly expensive controllers, but to provide especially useful time-oriented control mecanisms, thus allowing the user to focus on the *real added value* of the client-controller, or to write very easily simple perception-action loops directly in URBI.

URBI for Webots (Linux only)

URBI for Webots is URBI server for Webots, meaning that it is an application, running as a Webots controller, which acts as a URBI server, whereas the program which actually controls your simulated robot, is either a URBI script or a URBI client you have designed.

As of Webots 5.0.10, URBI for Webots is only implemented in the Linux version of Webots. Windows and Mac OS X versions of URBI for Webots are currently under development and will be integrated in upcoming versions of Webots.

¹http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/

²<http://uei.ensta.fr/eng/>

³<http://www.ensta.fr>

⁴<http://www.urbiforge.com>

In this tutorial we will only focus on the first possibility. Please refer to general URBI documentation to get familiar with the second one. It is strongly advised that you have a look to the URBI language documentation and tutorial available at URBI homepage.

The basic use of URBI for Webots rely on:

- The URBI for Webots controller is called `URBI.webots` and is stored in the `lib/urbi` subdirectory of Webots. Each Webots controller directory using URBI for Webots has a script file calling this executable.
- An XML configuration file giving some information specific to your robot, and especially declaring which devices are available, their names and parameters.
- A telnet connection on the port 54000 of the machine running URBI for Webots to send simple commands manually.

Installation of URBI for Webots (Linux only)

To install URBI for Webots on your Linux machine, you will have to obtain the latest version of the `webots-urbi-X.Y.Z.tar.bz2` package where X Y and Z represent version numbers and uncompress it in a local directory. This will create an `INSTALL` text file and a `webots` directory. Just copy the contents of the `webots` directory into your `webots` installation directory which is usually located in `/usr/local/`. You may also want to copy the `data`, `controllers` and `worlds` subdirectories in your local Webots user directory as well in order to be able to work on these files.

4.9.2 Running the examples

Simply open the `kiki_urbi.wbt` model in the Webots `worlds` directory, and click on the **play** button. The *kiki* robot should zoom across the place, avoiding walls: a small, very simple to understand and modify, URBI script is actually running!

An other example, more complex but fancy, is provided in the `aibo_ers7_urbi_dance.wbt` world: this script, written by Diego de Pardo, has won the Sony Daft-Punk dance competition on a real Aibo robot. Finally, another model is provided in the `aibo_ers7_urbi.wbt` world to serve as a basis for your own Aibo simulations.

4.9.3 Kiki and URBI

To learn more about URBI for Webots, let's have a closer look at this *kiki* robot programmed with an URBI script. First of all copy the following directories from the Webots installation directory to your Webots user directory:

- controllers/kiki_urbi
- data/urbi
- worlds/kiki_urbi.wbt

Open your local copy of `kiki_urbi.wbt` and check that it runs properly. Now open a telnet connection to `localhost` on port 54000 (the standart URBI port) and type the following:

```
kiki.stop(); // kiki should stop.
wheelL.val = 300 & wheelR.val = 250; // kiki should move in circle
stop toto; // To disable the watching of the wall
```

Try it and next time the robot gets close to a wall it will simply bump into it. Now type `stop wave` to stop it) and type the following:

```
wave: wheelL.val = 150 sin:3000 ampli:50 &
wave: wheelR.val = 150 sin:3000 ampli:50 phase:180,
```

4.9.4 Going further

Note: As you will experiment with URBI for Webots, keep in mind that when you **stop** the Webots simulation, it also suspends the associated controllers, hence it suspends the URBI server, so that your telnet session will not respond until you press **play** again to start the simulation.

To have a closer look at how URBI for Webots works, go to the `controllers/kiki_urbi` directory and open the following files with your favorite text editor:

- `kiki_urbi`: This is an executable shell script which calls the actual executable with the appropriate arguments.
- `URBI.INI`: This URBI script is launched at each start-up of the server from this directory. In this example, it contains an *alias* allowing you to use `wheels` to refering to both wheels, and it also loads the `tut1.u` demo script which is located in your `data/urbi/kiki` user directory.

You should also look into the `data/urbi/kiki` directory and open the following files with your favorite text editor:

- `kiki.xml`: This is the XML configuration file used by URBI for Webots: it holds both some information extracted from the model (i.e., the `kiki_urbi.wbt` file) and URBI-specific information such as the URBI names of the Webots devices, or the port number on which to listen.
- `tut1.u`: This is the main URBI script controlling the *kiki* robot.

You can now start to create your own scripts and load them from the `URBI.INI` file.

4.9.5 Making your own URBI for Webots controllers

As soon as you will want to go further, you will have to create your own URBI for Webots based controllers. Here is how to proceed:

1. Create a new directory in your `controllers` directory (say `toto`).
2. Copy into your `toto` directory the `kiki.urbi` script file and rename it to `toto` as it should exactly match the directory name.
3. Edit the `toto` script to possibly change the reference to the XML robot definition file and to the directory references where URBI scripts are searched.
4. Edit your world file and check that the controller of your robot is pointing to your `toto` controller (i.e., the `controller` field of your robot should be set to `toto`). Also, the `basicTimeStep` field of the `WorldInfo` node should be the same as the `frequency` attribute of the URBI for Webots XML tag. Finally, the `Battery` XML tag should be present and the `capacity` attribute should reflect the values indicated in the model. More generally, for each supported Webots device you want to be accessible from URBI, there should be a matching XML tag in one `Devices` section, and its attributes should reflect the same information as in the model.

4.9.6 Customizing and contributing to URBI for Webots

For now, URBI for Webots only supports the following devices:

- `Servo`: This is a standard, fully supported device.
- `DifferentialWheels`: Stricly speaking this is not a device but for pratical purpose this is of small importance: it is a robot from Webots point of view and a *hyper-device* from URBI for Webots point of view. But it creates two URBI devices (*wheelL* and *wheelR*) and is activated and parametrized in a similar way as a device.
- `DistanceSensor`: This is also a standard, fully supported device. You must only be careful regarding the unit policy (the URBI side value is proportionnal to the Webots side-value, it is up to you to decide which, if any, units you want to use).
- `Battery` This is not strictly speaking a device either, but it is fully supported: if you provide a *Battery* tag and the battery simulation is activated, then the URBI `power()` function will actually return the power level.

It is however easy to add support for other Webots devices or for your own *soft-devices* (i.e., software components), without having to edit the application code. You can do it by simply writing your own C++ piece of code and link it to the main part of the application. Please refer to the technical documentation of URBI for more information about this.

Chapter 5

Fast2D Simulation Mode

5.1 Introduction

In addition to the usual 3D and physics-based simulation modes, Webots offers a 2D simulation mode called Fast2D. The Fast2D mode enables very fast simulation for worlds that require only 2D computations. Many simulations are carried out on a 2D area using wheeled robots such as Alice™ or Khepera™; in such simulations the height and elevation of the objects is often irrelevant, therefore by using Fast2D, the overhead of 3D computations can be avoided. Typically Fast2D is designed for situations where the speed of a simulation is more important than its realism, as, for example, in evolutionary robotics or swarm intelligence.

5.2 Plugin Architecture

5.2.1 Overview

The Webots' Fast2D mode is built on a *plugin* architecture. The Fast2D plugin is a dynamically linked library that provides the functions necessary for the 2D simulation. These functions are responsible for the simulation of:

- Differential wheels robots (kinematics, friction model, collision detection)
- Obstacles (collision detection)
- Distance sensors (distance measurement)

The plugin architecture makes it possible to use different plugins for different worlds (.wbt file) and it also allows Webots users to design their own custom plugins.

5.2.2 Dynamically Linked Library

The Fast2D plugin is loaded by Webots when the user loads a world (.wbt file) that requires Fast2D simulation mode. The `WorldInfo` node of the world has a field called `fast2d` which specifies the name of the dynamically linked library to use as plugin for this world. For example:

```
WorldInfo {
  fast2d "enki"
}
```

An empty `fast2d` field means that no plugin is required and that the simulation must be carried out in 3D mode. When the `fast2d` field is not empty, Webots looks for a corresponding plugin in the `fast2d` directory located at the same directory level as the `worlds` and `controllers` directories. More precisely, Webots looks for the plugin at these two locations:

1. `$(userdir)/fast2d/$(pluginname)/$(pluginname).$(extension)`
2. `$(webotsdir)/fast2d/$(pluginname)/$(pluginname).$(extension)`

Where `$(userdir)` represents Webots' user directory as defined in the Webots preferences, where `$(pluginname)` is the plugin name as specified in the `fast2d` field of the `WorldInfo` node, where `$(extension)` is an operating system dependant filename extension such as `so` (Linux) or `dll` (Windows) and where `$(webotsdir)` is the path specified by the `WEBOTS_HOME` environment variable. If `WEBOTS_HOME` is undefined then `$(webotsdir)` is the path from where the Webots executable was started. If the required plugin is not found, Webots attempts to run the simulation using the built-in 3D simulator. According to the "enki" example above, and assuming that the user directory `$(userdir)` defined in the Webots preferences is `/home/user/webots` and that `WEBOTS_HOME=/usr/local/webots`, then the Linux version of Webots looks for the plugin in:

1. `/home/user/webots/fast2d/enki/enki.so`
2. `/usr/local/webots/webots/fast2d/enki/enki.so`

Since the plugin name is referred to by the `WorldInfo` node of a world (.wbt file), it is possible to have a different plugin for each world.

5.2.3 Enki Plugin

The Linux and Windows distributions of Webots come with a preinstalled Fast2D plugin called the *Enki plugin*. At the time of writing the Enki plugin is not available in the Mac OS X version of Webots. The Enki plugin is based on the Enki simulator, which is a fast open

source 2D robot simulator developed at the Laboratory of Intelligent Systems, EPFL in Lausanne (Switzerland) by Stephane Magnenat <stephane.magnenat@epfl.ch>, Markus Waibel <markus.waibel@epfl.ch> and Antoine Beyeler <antoine.beyeler@epfl.ch>. Please find more information about Enki at the LIS website¹.

5.3 How to Design a Fast2D Simulation

Webots' scene tree allows a large choice of 3D objects to be assembled in complex 3D worlds. Because Fast2D is designed to run simulations exclusively in 2D, the 3D worlds must be simplified before the Fast2D simulation can handle them properly.

5.3.1 3D to 2D

The most important simplification, is to remove one dimension from the 3D worlds; this is carried out by Webots automatically. In 3D mode, the xz-plane is traditionally used to represent the ground, while the positive y-axis represents the up direction. In Fast2D mode Webots projects the 3D objects onto the xz-plane simply by removing the y-dimension. Therefore, the Fast2D mode ignores the y-axis of the 3D and carries out simulation in the xz-plane exclusively. But then, the naming convention is to use the Fast2D y-axis to represent the 3D z-axis. See table 5.1.

3D	->	Fast2D
x	->	x
y	->	none
z	->	y
alpha (rotation angle)	->	-alpha (orientation angle)

Table 5.1: Conversion from 3D to Fast2D coordinate systems.

In short, the 3D y-axis does not matter with Fast2D. The objects' heights and elevations are ignored and therefore, the worlds intended for Fast2D simulation must be designed with that in mind. Furthermore, Fast2D worlds must also be designed such that the y-axes of all its Solid and DifferentialWheels nodes are aligned with the world's y-axis. In other words the rotation field of Solid and DifferentialWheels nodes must be:

```
Solid {
  rotation 0 1 0 <alpha>
  ...
}
```

Where alpha is the only parameter that you can tune. If a Fast2D world does not fulfil this requirement, the result of simulation is undefined. Note also that, the Fast2D (orientation) angles are equals to the negated 3D rotation angle. See table 5.1.

¹<http://lis.epfl.ch/>

5.3.2 Scene Tree Simplification

In Fast2D mode, Webots takes only the top level objects of the scene tree into account. Each Solid or DifferentialWheels node defined at the root level, will take part in the Fast2D simulation but the other Solid or DifferentialWheels nodes will be ignored. Still, it is possible to use a Solid as a child of another Solid or as a child of a DifferentialWheels but be aware that, in this case, although the child Solid does appear graphically, it is not taken into account by the simulation.

5.3.3 Bounding Objects

In Fast2D, just as in 3D simulation, only the objects' bounding objects are used in collision detection. Although Webots allows a full choice of bounding objects, in Fast2D mode, it is only possible to use a single Cylinder or a single Box as bounding object. Furthermore, the Fast2D mode requires that the coordinate systems of an object and of its corresponding bounding object must be the same. In other words, any Transform of the bounding object, will be ignored in Fast2D mode.

5.4 Developing Your Own Fast2D Plugin

The Enki based Fast2D plugin that comes with Webots is highly optimized and should be suitable for most 2D simulations. However in some cases you might want to use your own implementation of kinematics and collision detection. In such a case you will have to develop your own Fast2D plugin and this section explains how to proceed.

5.4.1 Header File

The types and interface required to compile your own Fast2D plugin are defined in the `fast2d.h` header file. This file is located in the Webots installation directory, in the `include/fast2d` subdirectory. It can be included like this:

```
#include <fast2d/fast2d.h>
...
```

The `fast2d.h` file contains C types and function declarations and therefore it can be compiled with either a C or C++ compiler.

5.4.2 Fast2D Plugin Types

Four basic types are defined in `fast2d.h`: `ObjectRef`, `SolidRef`, `RobotRef` and `SensorRef`. In order to enforce a minimal amount of type-checking and type-awareness, these basic types are

declared as non-interchangeable pointer types. They are only dummy types, not designed to be used as such, but rather to be placeholders for the real data types that the plugin programmer is responsible for implementing. So we suggest that you declare your own four data types as C structs or C++ classes. Then in your implementation of the Fast2D functions you should cast the addresses of your data instances onto the Fast2D types, as in the example below, where `MyRobotClass` and `MySensorClass` are user-defined types:

```
RobotRef webots_fast2d_create_robot() {
    return (RobotRef) new MyRobotClass();
}

void webots_fast2d_robot_add_sensor(RobotRef robotRef,
    SensorRef sensorRef, double x, double y, double angle) {

    MyRobotClass *robot = (MyRobotClass*) robotRef;
    MySensorClass *sensor = (MySensorClass*) sensorRef;
    robot->addSensor(sensor, x, y, angle);
    ...
}
```

In this example, Webots calls `webots_fast2d_create_robot()` when it requires a new robot object; this function instantiates the object and casts its address into a Fast2D type before returning it. After that, Webots will pass back this pointer as an argument to every subsequent plugin call that involves the same object. Apart from storing its address and passing it back, Webots does nothing with the object and then, it is completely safe for you to cast any pointer type. However, the simplest and most effective way for you to proceed, is to directly cast the addresses of your data instances, but you are free to do otherwise, provided that you assign a unique reference to each object.

Now, your data types should contain a certain number of attributes in order for the Fast2D functions to be able to operate on them. The UML diagram proposed in figure 5.1 shows the types and attributes that make sense according to the Fast2D functionality. This diagram is an implementation guideline for your own type declarations. We recommended implementing four data types in order to match exactly the four Fast2D basic types and we also suggest that in the implementation of these types you use similar attributes as those indicated in the diagram.

- `ObjectRef`: Reference to a solid or a robot object. `ObjectRef` is used in the Fast2D API to indicate that both `SolidRef` and `RobotRef` are suitable parameters. `ObjectRef` can be considered as a base class for a solid object or a robot because it groups the attributes common to both objects. These attributes are the object's position (`xpos` and `ypos`) and orientation (`angle`), the object's mass, the object's bounding radius (for circular objects) and the object's bounding rectangle (for rectangular objects). The object's position and angle are defined with respect to the world's coordinate system.
- `SolidRef`: Reference for a solid object. A `SolidRef` has the same physical properties as `ObjectRef`, but it is used to implement a wall or another obstacle.

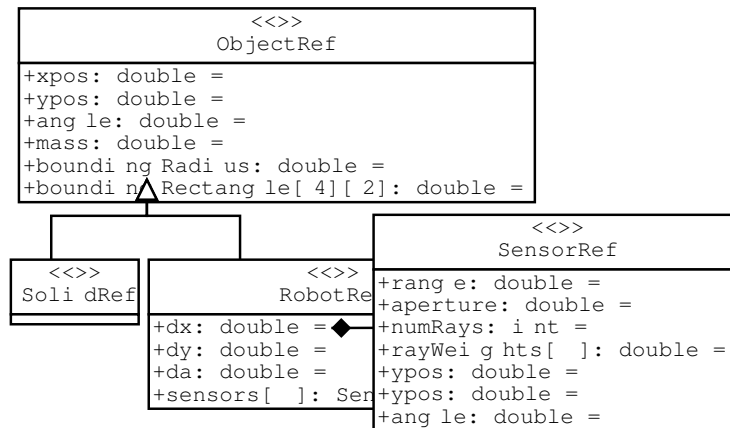


Figure 5.1: Fast2D Plugin Entity Relationship

- **RobotRef:** Reference for a robot object. A RobotRef has the same physical properties as ObjectRef, but in addition its speed (dx and dy) and angular speed (da) can be controlled. It is used to implement a differential wheel robot.
- **SensorRef:** Reference for a distance sensor object. A SensorRef represents a distance sensor that must be associated with a robot (RobotRef). SensorRef attributes are: the sensor's maximal range (range), the sensor's aperture angle [radian], the number of rays of the sensor (numRays), the weight of the individual rays (rayWeights), the positions (xpos and ypos) and orientation (angle) of the sensor. The sensor's position and angle are defined with respect to the coordinate system of the corresponding robot.

5.4.3 Fast2D Plugin Functions

In order for your plugin to be operational, it has to implement *all* the Fast2D functions. Once the plugin is loaded Webots checks that every function is present and if a single function is missing, Webots will attempt to run the simulation using the built-in 3D routines instead of the Fast2D plugin.

The Fast2D API uses two types of coordinates: *global* and *local*. The *global* coordinate system is the world's coordinate system, as described in table 5.1. Positions and angles of ObjectRef (including RobotRef and SolidRef) are expressed in the *global* coordinate system. In the other hand, the position and angle of SensorRef and the coordinates of bounding rectangles are expressed in the *local* coordinate system of the object they belong to. For example the position and angle of a sensor is expressed with respect to the local coordinate system of the robot which holds the sensor. As in 3D, in the Fast2D coordinate systems the zero-angle is aligned with the x-axis.

void webots_fast2d_init()

The `webots_fast2d_init()` function initializes the plugin. This function is called before any other Fast2D function: its purpose is to allocate and initialize the plugin's global data structures. Note that when the **Revert** button is pressed or whenever anything changes in the scene tree, Webots reinitializes the plugin by calling first `webots_fast2d_cleanup()` and then `webots_fast2d_init()`. See also figure 5.2.

void webots_fast2d_cleanup()

Free all the data structures used by the plugin. After a call to this function no further Fast2D call will be carried out by Webots, with the exception of `webots_fast2d_init()`, because at this moment the plugin data structures are supposed to have been deleted. A subsequent call to `webots_fast2d_init()` will indicate that the plugin must be reinitialized because the world is being redefined. The plugin is responsible for allocating and freeing every Fast2D objects. If `webots_fast2d_cleanup()` fails to free all the memory that was allocated by the plugin, this will result in memory leaks in Webots.

void webots_fast2d_step(double dt)

Run a simulation step of `dt` seconds. This function is invoked by Webots once for each simulation step (basic simulation step), for example, during **Play** or once each time the **Step** button is pressed. The `dt` parameter corresponds to the world's basic time step (of the `WorldInfo` node) converted to seconds (i.e., divided by 1000). The responsibility of this function is to compute the new position and angle (as returned by `webots_fast2d_object_get_transform()`) of every simulated object (`ObjectRef`) according to your implementation of kinematics and collision handling. This is usually one of the functions that requires the largest amount of work on your side.

RobotRef webots_fast2d_create_robot()

Request the creation of a robot in the plugin. This function must return a valid robot reference (`RobotRef`) to Webots. The exact properties of the robot will be specified in subsequent Fast2D calls.

SolidRef webots_fast2d_create_solid()

Request the creation of a solid object in the plugin. This function must return a valid solid reference (`SolidRef`) to Webots. The exact properties of the solid object will be specified in subsequent Fast2D calls.

void webots_fast2d_add_object(ObjectRef object)

Request the insertion of an object (robot or solid) into the 2D world model. This function is called by Webots after an object's properties have been set and before executing the first simulation step (`webots_fast2d_step()`).

SensorRef webots_fast2d_create_irsensor(double range, double aperture, int numRays, const double rayWeights[])

Requests the creation of an infra-red sensor. This function must return a valid sensor reference (`SensorRef`) to Webots. The `range` parameter indicates the maximal range of the sensor. It is determined according to the `lookupTable` of the corresponding `DistanceSensor` in the Webots scene tree. The `aperture` parameter corresponds to the value of the `aperture` field of the `DistanceSensor`. The `numRays` parameter indicates the value of the `numberOfRays` field of the `DistanceSensor`. The `rayWeights` parameter is an array of `numRays` double numbers that specifies the individual weights that must be associated with each sensor ray. The sum of the ray weights provided by Webots is always exactly 1.0, and it is always left/right symmetrical. For more information on the sensor weights, please refer to the description of the `DistanceSensor` node in Webots Reference Manual. In order to be consistent with the Webots graphical representation, the plugin's implementation of the sensors requires that:

- All the rays have the same length (the specified sensor range)
- The rays are distributed uniformly (equal angles from each other)
- The angle between the first and the last ray equals exactly to the specified aperture

void webots_fast2d_robot_add_sensor(RobotRef robot, SensorRef sensor, double xpos, double ypos, double angle)

Add a sensor to a robot. The `robot` parameter is a robot reference previously created through `webots_fast2d_create_robot()`. The `sensor` parameter is a sensor reference previously created through `webots_fast2d_create_irsensor()`. The `xpos`, `ypos` and `angle` parameters indicate the desired position and orientation of the sensor in the the local coordinate system of the robot.

double webots_fast2d_sensor_get_activation(SensorRef sensor)

Request to return the current distance measured by a sensor. The `sensor` parameter is a sensor reference that was created through `webots_fast2d_create_irsensor()`. This function must return the average of the weighted distances measured by the sensor rays. The distances must be weighted using the `rayWeights` values that were passed to `webots_fast2d_create_`

`irsensor()`. Note that this function is responsible only for calculating the *weighted average distance* measured by the sensor. It is Webots responsibility to compute the *final activation value* (the value that will finally be returned to the controller) from the average distance and according to the `DistanceSensor`'s lookup table.

`void webots_fast2d_object_set_bounding_rectangle(ObjectRef object, const double x[4], const double y[4])`

Define an object as rectangular and set the object's bounding rectangle. The `object` parameter is a solid or robot reference. The `x` and `y` arrays specify the coordinates of the four corners of the bounding rectangle in the object's coordinate system. The sequence $(x[0], y[0])$, $(x[1], y[1])$, $(x[2], y[2])$, $(x[3], y[3])$ is specified counter-clockwise.

`void webots_fast2d_object_set_bounding_radius(ObjectRef object, double radius)`

Define an object as circular and set the object's bounding radius. The `object` parameter is a solid or robot reference. In the Fast2D plugin, an object can be either rectangular or circular; Webots indicates this by calling either `webots_fast2d_object_set_bounding_rectangle()` or `webots_fast2d_object_set_bounding_radius()`.

`void webots_fast2d_object_set_mass(ObjectRef object, double mass)`

Defines the mass of an object. The `object` parameter is a solid or robot reference. The `mass` parameter is the object's required mass. According to your custom implementation the mass of an object can be involved in the calculation of a robot's acceleration and ability to push other objects. The implementation of this function is optional. Note that Webots calls this function only if the corresponding object has a `Physics` node. In this case the `mass` parameter equals the `mass` field of the `Physics` node. A negative mass must be considered infinite. If your model does not support the concept of mass, you should implement an empty `webots_fast2d_object_set_mass()` function.

`void webots_fast2d_object_set_position(ObjectRef object, double xpos, double ypos)`

Request to set the position of an object. The `object` parameter is a solid or robot reference. The `xpos` and `ypos` parameters represent the required position specified in the global coordinate system. This function is called by Webots during the construction of the world model. Afterwards, the object positions are only modified by the `webots_fast2d_step()` function. See also figure 5.2.

void webots_fast2d_object_set_angle(ObjectRef object, double angle)

Request to set the angle of an object. The `object` parameter is a solid or robot reference. The `angle` parameter is the requested object's angle specified in the global coordinate system. This function is called by Webots during the construction of the world model. Afterwards, the object angles are only modified by the `webots_fast2d_step()` function. See also figure 5.2.

void webots_fast2d_robot_set_speed(RobotRef robot, double dx, double dy)

Request to change the speed of a robot. The `robot` parameter is a robot reference. The `dx` and `dy` parameters are the two vector components of the robot's speed in the global coordinate system. This corresponds to a change of the position of the robot (`xpos` and `ypos`) per second. More precisely: $dx = v * \sin(\alpha)$ and $dy = v * \cos(\alpha)$, where α is the robot's orientation angle and where v is the absolute robot's speed which is calculated according to the wheels radius and wheels rotation speed. For more information, see the description of the `DifferentialWheels` node and the `differential_wheels_set_speed()` in Webots Reference Manual.

void webots_fast2d_robot_set_angular_speed(RobotRef robot, double da)

Request to change the angular speed of a robot. The `robot` parameter is a robot reference. The `da` parameter indicates the requested angular speed. A robot's angular speed is the speed of its rotation around its center in radian per second.

void webots_fast2d_object_get_transform(ObjectRef object, double *xpos, double *ypos, double *angle)

Read the current position and angle of an object. The `object` parameter is a robot or solid reference. The `xpos`, `ypos` and `angle` parameters are the addresses of double numbers where this function must write the values. These parameters are specified according to the global coordinate system.

5.4.4 Fast2D Plugin Execution Scheme

This section describes the sequence used by Webots for calling the plugin functions. Please refer to the diagram of figure 5.2.

1. The plugin is loaded. Go to step 2.
2. The `webots_fast2d_init()` function is called. Go to step 3 or 5.

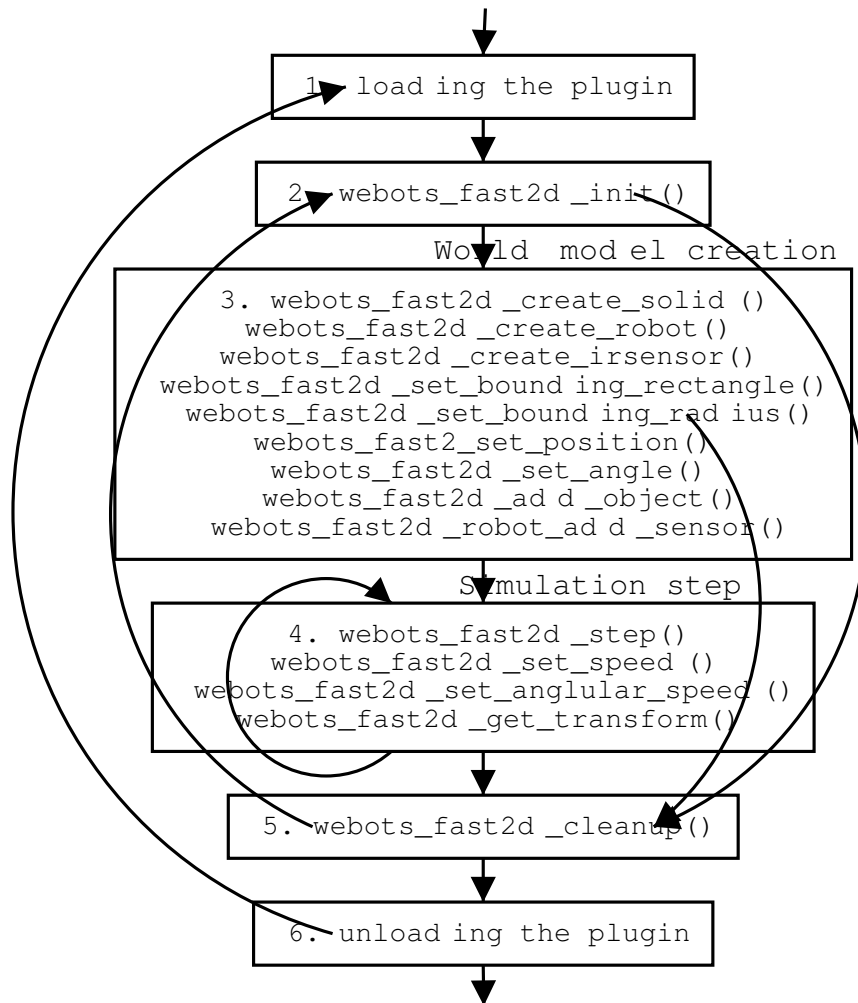


Figure 5.2: Fast2D Plugin Execution Scheme

3. The world model is created. This is achieved through a sequence of calls to the functions `webots_fast2d.create_*`(), `webots_fast2d.set_*`() and `webots_fast2d.add_*`(). Question marks are used to represent a choice among several functions names. Although the exact sequence is unspecified, for each object it is guaranteed that: the corresponding `webots_fast2d.create_*`() function is called first, the corresponding `webots_fast2d.set_*`() functions are called next and that the corresponding `webots_fast2d.add_*`() function is called last. Go to step 4 or 5.
4. A simulation step is carried out. This is achieved through an unspecified sequence of calls to `webots_fast2d.step()`, `webots_fast2d.set_speed`, `webots_fast2d.set_angular_speed()` and `webots_fast2d.get_transform()`. Go to step 4 or 5.
5. The `webots_fast2d.cleanup()` function is called. Go to step 2 or 6.
6. The plugin is unloaded. Go to step 1.

5.4.5 Fast2D Execution Example

This last section shows an example of a Webots scene tree and the corresponding Fast2D calls that are carried out in case the world is interpreted using Fast2D. Suspension marks represent omitted code or parameters. By examining this example carefully, you will notice that, as it was explained earlier, when transformed from 3D to Fast2D:

- The objects rotation angles are negated
- The objects y-coordinates (height and elevation) are ignored
- The 3D z-axis becomes the Fast2D y-axis

```

Solid {
  translation 0.177532 0.03 0.209856
  rotation 0 1 0 0.785398
  ...
  boundingObject Box {
    size 0.2 0.06 0.2
  }
}

DifferentialWheels {
  translation -0.150197 0 0.01018
  rotation 0 1 0 -4.85101
  children [
    ...
    DistanceSensor {
      translation -0.0245 0.0145 -0.012
      rotation 0 1 0 3.0543
      ...
      lookupTable [
        0 1023 0
        0.05 0 0.01
      ]
      aperture 0.5
    }
  ]
  ...
  boundingObject Transform {
    translation 0 0.011 0
    children [
      Cylinder {
        height 0.022
        radius 0.0285
      }
    ]
  }
}

```



```
        }
    ]
}
...
}

webots_fast2d_init()
webots_fast2d_create_solid()
webots_fast2d_object_set_bounding_polygon(...)
webots_fast2d_object_set_position(..., xpos=0.177532, ypos=0.209856)
webots_fast2d_object_set_angle(..., angle=-0.785398)
webots_fast2d_add_object()

webots_fast2d_create_robot()
webots_fast2d_object_set_bounding_radius(..., radius=0.0285)
webots_fast2d_object_set_position(..., xpos=-0.150197, ypos=0.01018)
webots_fast2d_object_set_angle(..., angle=4.85101)
webots_fast2d_add_object()

webots_create_irsensor(range=0.05, aperture=0.5, numRays=1, ...)
webots_fast2d_robot_add_sensor(..., xpos=-0.0245, ypos=-0.012, angle=-3.0543)
```

Finally, note that the largest input value of the `DistanceSensor`'s lookup table (0.05), becomes the sensor's range in `Fast2D`.

You will find further information about the `DifferentialWheels` and `DistanceSensor` nodes and controller API in `Webots Reference Manual`.

Chapter 6

Using the KheperaTM robot

The goal of this chapter is to explain how to use Webots with your Khepera robot. Khepera is a mini mobile robot developed by K-Team SA, Switzerland (www.k-team.com).

Webots can use the serial port of your computer to communicate with the Khepera robot.

6.1 Hardware configuration

1. Configure your Khepera robot in mode 1, for serial communication protocol at 9600 baud as described in figure 6.1.
2. Plug the serial connection cable between your Khepera robot and the Khepera interface.
3. Plug the Khepera Interface into a serial port of your computer (either COM1 or COM2, at your convenience).
4. Check the the Khepera robot power switch is OFF and plug the power supply to the Khepera Interface.

Note: Linux and Mac OS X users may want to redefine the COM1, COM2, COM3 and COM4 ports by setting `WEBOTS_COM1`, `WEBOTS_COM2`, `WEBOTS_COM3` and/or `WEBOTS_COM4` environment variables to point to the appropriate `/dev` device files.

On Linux, if these environment variables are not set, Webots will use respectively `/dev/ttyS0`, `/dev/ttyS1`, `/dev/ttyS2` and `/dev/ttyS3` for COM1, COM2, COM3 and COM4 (note the -1 difference). For example, if your laptop running Linux has no serial port, you may want to use a USB-RS232 converter, in which case it may be useful to type something like: `export WEBOTS_COM1 /dev/ttyUSB0` to allow Webots to communicate with the Khepera through the USB port.

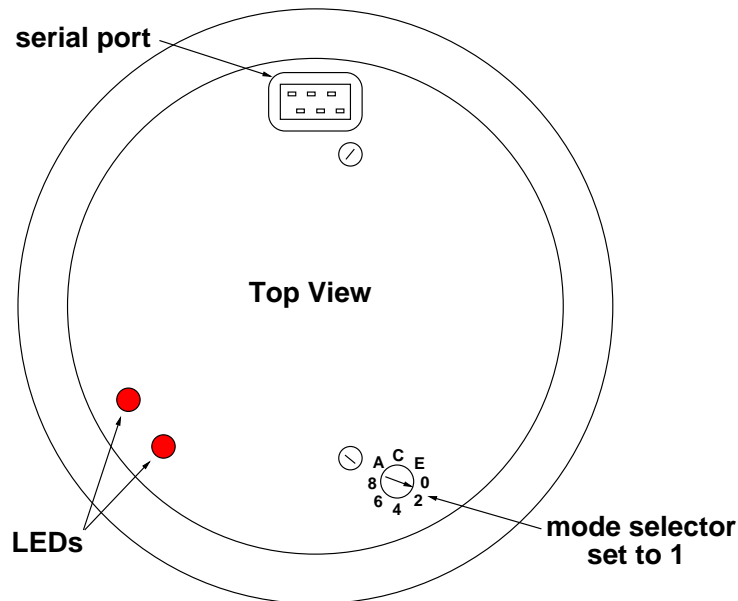


Figure 6.1: Khepera II mode selection

On Mac OS X, only COM1 has a default value which is set to `"/dev/tty.USB Serial"`, corresponding to the default USB to serial converter (like the one installed by the USB232-P9 converters). Other USB to serial converters may require that you define the `WEBOTS_COM1` environment variable to match their specific value. For example, the KeySpan USB to serial converter will need that you define `WEBOTS_COM1` as `"/dev/tty.USA28X1213P1.1"`. Please consult the documentation of your USB serial adapter to know the exact file name to be defined.

That's it. Your system is operational: you will now be able to simulate, remote control and transfer controllers to your Khepera robot.

6.2 Running the simulation

Launch Webots: on Windows, double click on the lady bug icon, on Linux, type `webots` in a terminal. Go to the **File Open** menu item and open the file named `khepera.wbt`, which contains a model of a Khepera robot (see figure 6.2) associated with a Khepera controller (see figure 6.3). If the Khepera controller window do not show up, press the **Step** button in the main window of Webots.

You can navigate in the scene using the mouse pointer. To rotate the scene, click on the left button and drag the mouse. To translate the scene, use the right button. To zoom and tilt, use the middle button. You may also use the mouse wheel to zoom in or out.

Using these controls, try to find a good view of the Khepera robot. You have probably noticed that clicking on an object in the scene would select it. Select the Khepera robot and choose the

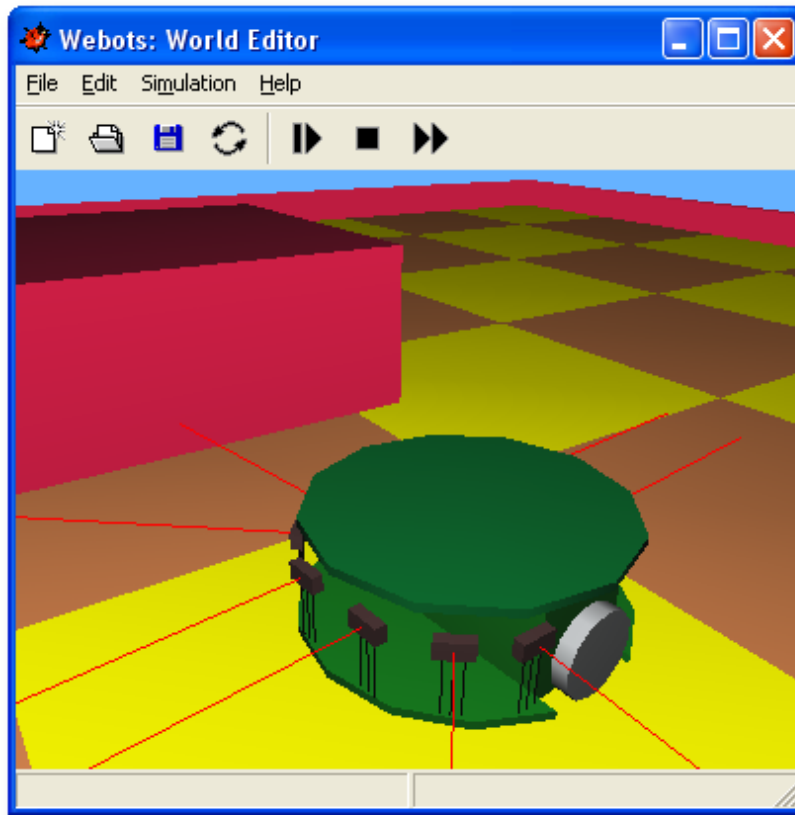


Figure 6.2: Khepera example world

Simulation Robot View menu item. This way, the camera will follow the robot moves. Then, click on the **Run** button to start up the simulation. You will see the robot moving, while avoiding obstacles.

To visualize the range of the infra red distance sensors, go to the **File Preferences...** menu item to pop up the Preferences window. Then, check the **Display sensor rays** check box in the **Rendering** tab.

In the controller windows, the values of the infra-red distance sensors are displayed in blue, while the light measurement values are displayed in light green. You can also observe the speed of each motor, displayed in red and the incremental encoder values displayed in dark green (see figure 6.3).

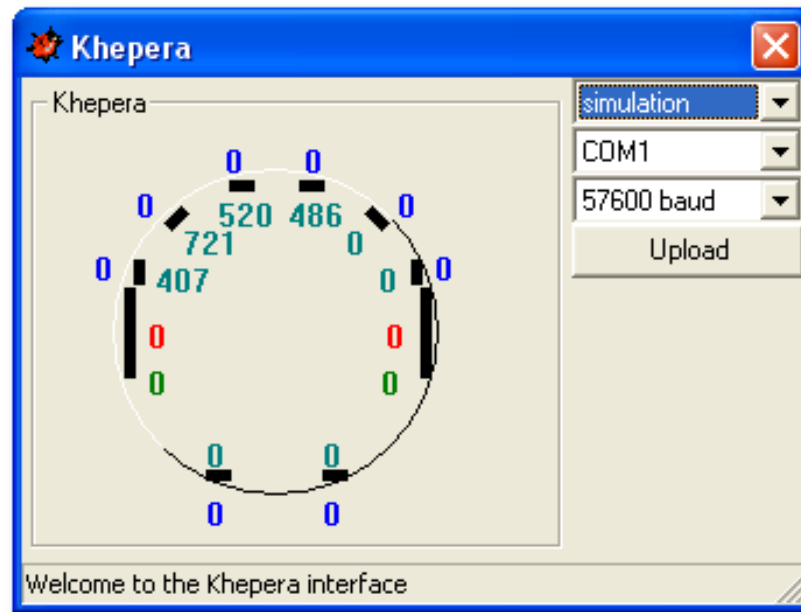


Figure 6.3: Khepera Controls

6.3 Understanding the model

6.3.1 The 3D scene

In order to better understand what is going on with this simulation, let's take a closer look at the scene structure. Double click on an object in the scene, or select the **Edit Scene Tree Window** to open the scene tree window. If you double clicked on an object, you will see that object selected in the scene tree (see figure 6.4). Clicking on the little cross icon of an object name in the scene tree, will expand that object, displaying its properties.

We will not describe in details the Webots scene structure in this chapter. It is build as an extension of the VRML97 standard. For a more complete description, please refer to the Webots user guide and reference manuals. However, let's have a first overview.

You can see that the scene contains several objects, which we call nodes. You can play around with the nodes, expanding them to look into their fields, and possibly change some values. The `WorldInfo` node contains some text description about the world. The `Viewpoint` node defines the camera from which the scene is viewed. The `Background` node defines the color of the background of the scene which is blue in this world. The `PointLight` node defines a light which is visible from the light sensors of the robot. The light location can be displayed in the scene by checking **Display Lights** in the **Rendering** tab of the preferences window. The remaining nodes are physical objects and have a `DEF` name for helping identifying them.

The `GROUND Transform` is not a `Solid` which means no collision detection is performed

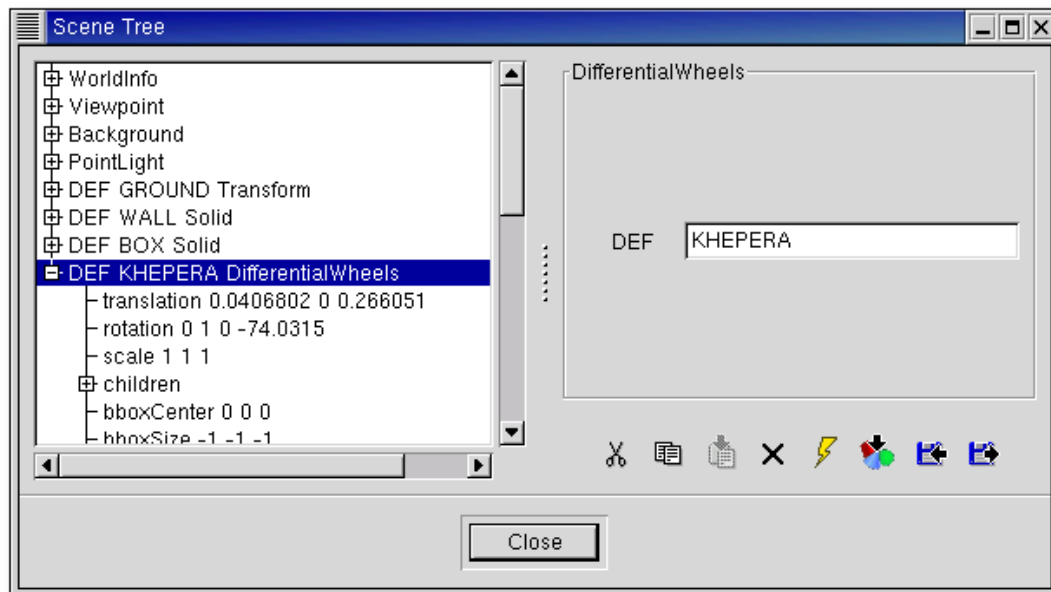


Figure 6.4: Scene tree window for the Khepera world

with this node. On the other hand, the WALL and BOX nodes are Solid nodes. They have a boundingObject field used for collision detection.

Finally, the KHEPERA DifferentialWheels node defines the Khepera robot.

6.3.2 The Khepera model

As you can guess, a DifferentialWheels node defines any differentially wheeled robot. The parameters provided here correspond to the size and functionalities of a Khepera robot. For example, if you expand the children list, you will be able to find some shapes defining the body of the robot and a number of sensors, including distance and light sensors. Although on the Khepera robot, the light and distance sensors are the same device, they are divided into two logical devices in the Webots model. This makes the simulator more modular and generic. Moreover, you will notice that each device (DifferentialWheels, DistanceSensor, LightSensor, etc.) has a list of children defining either sub devices or 3D shapes.

Webots recognizes this DifferentialWheels as a Khepera robot because its model field is set to "Khepera". Moreover, each sensor is named in a specific way in order to be recognized by Webots. For example, the distance sensor with a name set to "ds0" corresponds to the first infrared distance sensor. The Khepera interface recognized distance sensors named "ds0" to "ds7", light sensors named "ls0" to "ls7", camera sensor named "k213", and distance sensors named "fs0" to "fs2" (optional floor color sensors). This allows Webots to display the Khepera window when you double-click on the Khepera robot in the 3D world or when you choose the **Show Robot Window** menu item in the **Simulation** menu while the corresponding robot is selected.

The differential wheels model

The differential wheels model of a robot is defined by a number of parameters, including the axle length, the wheel radius, the maximum speed, maximum acceleration, the speed unit, slip noise and encoder noise. Values for these parameters are provided in this example to match approximately a Khepera robot. You may need to refine them if you need a very precise model. Please refer to the Webots user guide for a complete description of these parameters.

The sensor model

The distance sensors are simulated by computing the collision between a single sensor ray and objects in the scene. The response of the sensor is computed according to its `lookupTable` and modulated by the color of the object (since these sensors are of "infra-red" type, red objects are seen better than green ones). The `lookupTable` is actually a table of floating point values which is extrapolated to compute the response of the sensor. The first value is the distance expressed in meters (increasing the biggest distance value will make the sensor look further). The second value is the response read by the controller of the robot and the third value is the percentage of white noise associated to the distance and response, expressed in the range [0;1]. For a more complete discussion on the distance sensor model, please refer to the Webots user guide.

Light sensors are pretty similar to distance sensors. They also rely on a `lookupTable` for computing their return value according the measured value.

6.4 Programming the Khepera robot

6.4.1 The controller program

Among the fields of a `DifferentialWheels` node, you may have noticed the `controller` field. This field defines an executable program that will control the robot. By default executable programs are searched in the Webots `controllers` directory, but you can define another location in the Preferences **Files and paths** tab, under the **User path:** label. This path define a directory in webots will look for a `worlds` and a `controllers` directory. The `controllers` directory should contain subdirectories named after the names of the controllers (i.e., `khepera` in our case). This `khepera` directory should contain an executable file named `khepera.exe` on Windows or `khepera` on Linux. Moreover, along with the executable file, you will also find sources files and possibly makefiles or project files used to build the executable from the sources.

6.4.2 Looking at the source code

The source code of the example controller is located in the following file under the Webots directory:

controllers/khepera/khepera.c

It contains the following code:

```
#include <stdio.h>
#include <device/robot.h>
#include <device/differential_wheels.h>
#include <device/distance_sensor.h>
#include <device/light_sensor.h>

#define FORWARD_SPEED 8
#define TURN_SPEED 5
#define SENSOR_THRESHOLD 40

DeviceTag ds1,ds2,ds3,ds4,ls2,ls3;

void reset(void) {
    ds1 = robot_get_device("ds1"); /* distance sensors */
    ds2 = robot_get_device("ds2");
    ds3 = robot_get_device("ds3");
    ds4 = robot_get_device("ds4");
    ls2 = robot_get_device("ls2"); /* light sensors */
    ls3 = robot_get_device("ls3");
}

int main() {
    short left_speed=0,right_speed=0;
    unsigned short ds1_value,ds2_value,ds3_value,ds4_value,
                  ls2_value,ls3_value;
    int left_encoder,right_encoder;

    robot_live(reset);
    distance_sensor_enable(ds1,64);
    distance_sensor_enable(ds2,64);
    distance_sensor_enable(ds3,64);
    distance_sensor_enable(ds4,64);
    light_sensor_enable(ls2,64);
    light_sensor_enable(ls3,64);
    differential_wheels_enable_encoders(64);
    for(;;) { /* The robot never dies! */
        ds1_value = distance_sensor_get_value(ds1);
        ds2_value = distance_sensor_get_value(ds2);
        ds3_value = distance_sensor_get_value(ds3);
        ds4_value = distance_sensor_get_value(ds4);
        ls2_value = light_sensor_get_value(ls2);
        ls3_value = light_sensor_get_value(ls3);
```

```

if (ds2_value>SENSOR_THRESHOLD &&
    ds3_value>SENSOR_THRESHOLD) {
    left_speed = -TURN_SPEED; /* go backward */
    right_speed = -TURN_SPEED;
}
else if (ds1_value<SENSOR_THRESHOLD &&
        ds2_value<SENSOR_THRESHOLD &&
        ds3_value<SENSOR_THRESHOLD &&
        ds4_value<SENSOR_THRESHOLD) {
    left_speed = FORWARD_SPEED; /* go forward */
    right_speed = FORWARD_SPEED;
}
else if (ds3_value>SENSOR_THRESHOLD ||
        ds4_value>SENSOR_THRESHOLD) {
    left_speed = -TURN_SPEED; /* turn left */
    right_speed = TURN_SPEED;
}
if (ds1_value>SENSOR_THRESHOLD ||
    ds2_value>SENSOR_THRESHOLD) {
    right_speed=-TURN_SPEED; /* turn right */
    left_speed=TURN_SPEED;
}
left_encoder = differential_wheels_get_left_encoder();
right_encoder = differential_wheels_get_right_encoder();
if (left_encoder>9000)
    differential_wheels_set_encoders(0,right_encoder);
if (right_encoder>1000)
    differential_wheels_set_encoders(left_encoder,0);
/* Set the motor speeds */
differential_wheels_set_speed(left_speed,right_speed);
robot_step(64); /* run one step */
}
return 0;
}

```

This program is made up of two functions: a main function, as in any C program and function named `reset` which is a callback function used for getting references to the sensors of the robot. A number of includes are necessary to use the different devices of the robot, including the differential wheels basis itself.

The main function starts up by initializing the library by calling the `khepera_live` function, passing as an argument a pointer to the `reset` function declared earlier. This `reset` function will be called each time it is necessary to read or reread the references to the devices, called device tags. The device tag names, like "ds1", "ds2", etc. refer to the name fields you can see in the scene tree window for each device. The `reset` function will be called the first time from the

`khepera_live` function. So, from there, you can assume that the device tag values have been assigned.

Then, it is necessary to enable the sensor measurements we will need. The second parameter of the enable functions specifies the interval between updates for the sensor in millisecond. That is, in this example, all sensor measurements will be performed each 64 ms.

Finally, the main function enters an endless loop in which the sensor values are read, the motor speeds are computed according to the sensor values and assigned to the motors, and the encoders are read and sometimes reset (although this make no special sense in this example). Please note the `robot_step` function at the end of the loop which takes a number of milliseconds as an argument. This function tells the simulator to run the simulation for the specified amount of time. It is necessary to include this function call, otherwise, the simulation may get frozen.

6.4.3 Compiling the controller

To compile this source code and obtain an executable file, a different procedure is necessary depending on your development environment. On Linux, simply go to the controller directory where the `khepera.c` resides, and type `make`. On Windows, you may do exactly the same if you are working with Cygwin. If you use Dev-C++ or Microsoft Visual C++, you will need to create a project file and compile your program from your Integrated Development Environment. Template project files for both Dev-C++ and Visual C++ are available in the `braiten` controller directory.

Once compiled, reload the world in Webots using the **Revert** button (or relaunch Webots) and you will see your freshly compiled run in Webots.

6.5 Transferring to the real robot

6.5.1 Remote control

The remote control mode consists in redirecting the inputs and outputs of your controller to a real Khepera robot using the Khepera serial protocol. Hence your controller is still running on your computer, but instead of communicating with the simulated model of the robot, it communicates with the real device via connected to the serial port.

To switch to the remote control mode, your robot needs to be connected to your computer as described in section 6.1. In the robot controller window, select the **COM** popup menu corresponding to the serial port to which your robot is connected. Then, just click on the **simulation** popup menu in the controller window and select **remote control** instead. After a few seconds, you should see your Khepera moving around, executing the commands sent by your controller. The controller window now displays the sensor and motor values of the real Khepera.

You may press the simulation **stop** to stop the real robot. The **run** will restart it. The **step** button is helpful to run the real robot step by step. To return to the simulation mode, just use the popup menu previously used to start the remote control mode. You may remark that it is possible to change the baud rate for communicating with the robot. The default value is 57600 baud, but you may choose another value from the popup menu.

Important: If you change the baud rate with the popup menu, don't change the mode on the Khepera robot, since the baud rate is changed by software. The mode on the Khepera robot should always remain set to 1 (i.e., serial protocol at 9600 bauds).

6.5.2 Cross-compilation and upload

We assume in this subsection, that you have installed the `webots-kros` package provided with Webots.

Cross-compiling a controller program creates an executable file for the Khepera micro-controller from your C source file. In order to produce such an executable, you can use the `make` command either with the `Makefile.kros` file (for the Khepera robot) or the `Makefile.kros2` file (for the Khepera II robot). These files are provided within the `khepera` controller directory. From Linux, just type:

```
make -f Makefile.kros
```

for Khepera, or:

```
make -f Makefile.kros2
```

for Khepera II.

From Windows, launch the Webots-kros application and follow the instructions. In both cases you see the following messages telling you that the compilation is progressing successfully:

```
Compiling  khepera.c into khepera.s
Assembling khepera.s into khepera.o
Linking    khepera.o into khepera.s37
khepera.s37 is ready for Khepera (II) upload
```

It may be necessary to remove any previous `khepera.o` which may conflict with the one generated by the cross-compiler. In order to do so, you can type:

```
make -f Makefile.kros clean
```

Finally, to upload the resulting `khepera.s37` executable file onto the Khepera robot, click on the **upload** button in the controller window. Please note that you don't need to change the mode of the Khepera robot since the upload mode is activated by software from the mode 1. The green LED of your Khepera should switch on while uploading the program. It lasts for a few seconds or minutes before completing the upload. Once complete, the robot automatically executes the new program.

6.6 Working extension turrets

6.6.1 The K213 linear vision turret

The example world `khepera_k213.wbt` contains a complete working case for the K213 linear vision turret. The principles are the same as for the simple Khepera example, except that additional functions are used for enabling and reading the pixels from the camera. The function `camera_get_image` returns an array of unsigned characters representing the image. The macro `camera_image_get_grey` is used to retrieve the value of each pixel. As seen on figure 6.5, the camera image is displayed in the controller window as grey levels and as an histogram.

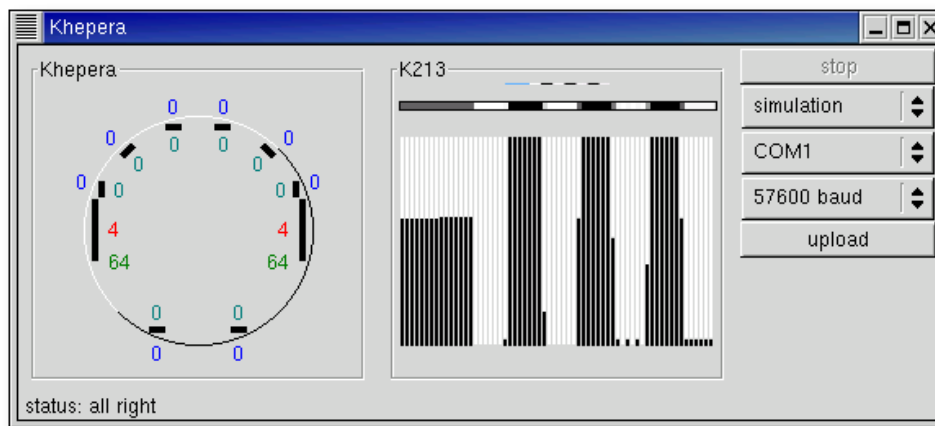


Figure 6.5: Khepera K213 controls

6.6.2 The Gripper turret

figure 6.6 shows the `khepera_gripper.wbt` example. In this example a model of a Khepera is equipped with a Gripper device. It can grab red cylinders, carry them away and put them down. From a modeling point of view, the Gripper turret is made up of two Webots devices:

- A *Servo* node which represents the servo motor controlling the height of the gripper (rotation).
- A *Gripper* node which represents the gripping device: the two fingers.

These devices can be configured to match more precisely the real one or to try new designs. For example, it is possible to configure the maximum speed and acceleration of the *Servo* node, simply by changing the corresponding fields of that node in the scene tree window.

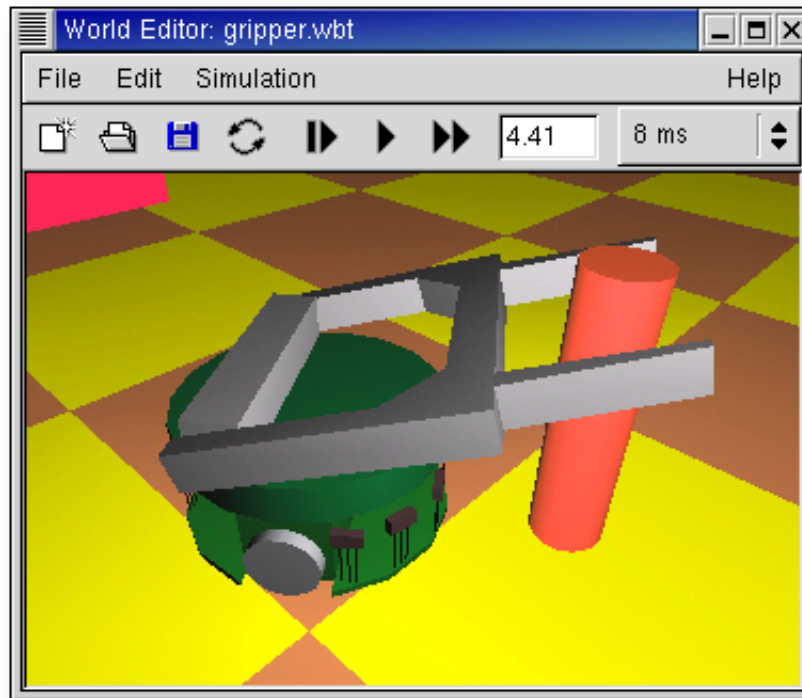


Figure 6.6: Khepera Gripper

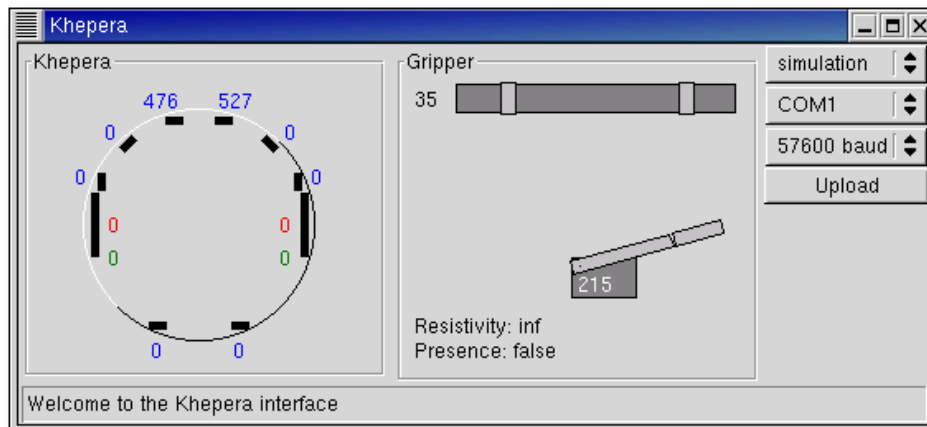


Figure 6.7: Khepera gripper controls

When clicking on a Khepera robot equipped with a gripper turret. The Khepera window popping up shows the gripper device (see figure 6.7). It shows the rotation of the gripper arm, the aperture of the grips, the presence of an object within the grips and the resistivity of a gripper object. If you have a real gripper mounted on a Khepera robot, it can be remote controlled by Webots.

6.6.3 Custom turrets and Khepera protocol

Webots offers the capability to communicate with the real Khepera robot from your controller program by using the standard Khepera communication protocol (see the Khepera manual for details about this protocol). The principle is simple: the Khepera robot defined in the `khepera.wbt` file has an emitter and a receiver device. The emitter is named `rs232_out` while the receiver is named `rs232_in`. You can send messages through the emitter, like `"B\n"` and retrieve the answer from the remote controller Khepera through the receiver which should be something like `"b,5.02,5.01"`, depending on the software version running on your Khepera robot. This will work only in remote control mode, not in simulation mode or in cross-compilation mode. It is especially useful if you have a custom extension turret on the top of your Khepera robot (use the `"T"` command), if you want to read the A/D inputs of the real robot (use the `"I"` command), or if you want to access any other command available in the Khepera protocol. An example of using this system is provided within the `khepera_serial.c` file which lies in the `khepera` directory of the Webots `controllers` directory.

6.7 Support for other K-Team robots

6.7.1 Koala™

The Webots distribution contains an example world with a model of a Koala robot. This robot is much bigger than the Khepera and has 16 infra-red sensors, as seen on figure 6.8. The example can be found in `worlds/koala.wbt`.

6.7.2 Alice™

An example of Alice robot is also provided. Alice is much smaller than Khepera and has two to four infra-red sensors. In our example, we have only two infra-red sensors (see figure 6.9). The example can be found in `worlds/alice.wbt`.

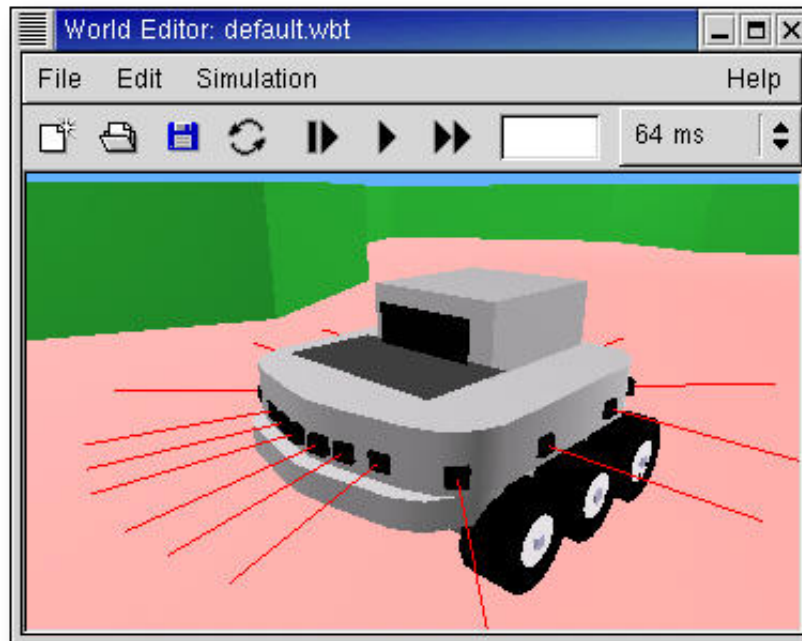


Figure 6.8: The Koala robot

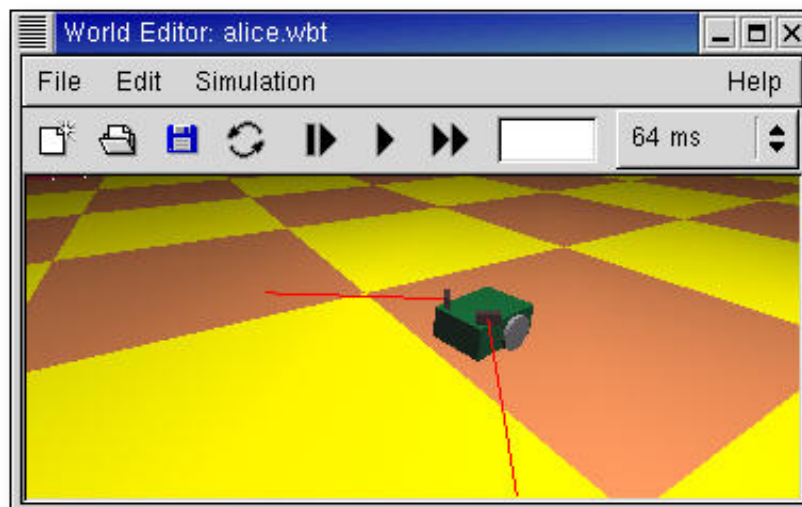


Figure 6.9: The Alice robot

Chapter 7

Using the LEGO Mindstorms™ robots

In this chapter, you will learn how to use Webots with the LEGO™ Mindstorms™ robots. The LEGO™ Mindstorms™ is a series of LEGO™ products allowing to build robots from LEGO™ bricks. A special brick called RCX is used to control the robot. This brick contains a micro-controller chip, a LCD display, a buzzer, 3 sensor inputs and 3 actuator outputs. Available sensors include touch sensors, light sensors, rotation sensors, temperature sensors. Actuators include motors and lights. The basic box, called "Robotics Invention System" includes two motors, two touch sensors and one light sensor. This chapter will be based on this basic box. However, Webots is not limited to this basic box and you could easily go beyond this chapter by creating much more complex virtual robots based on advanced LEGO™ Mindstorms™ elements.

The first section describes step by step instructions to build up the Rover robot. This robot will be used throughout this tutorial.

The second section describes the Webots model corresponding to the Rover robot. It explains how to program its controller in Java and how to compile it.

Finally, the last section explains how to cross-compile the Java controller you used for simulating the Rover in Webots. Once cross-compiled, your controller can be uploaded into a real Rover robot!

7.1 Building up the Rover robot

One of the most interesting model that can be build straight out the "Robotics Invention System" box is the Rover robot. This robot is described in this section. It has a two differential wheels drive system, a light sensor looking down to the ground and two touch sensors.

The following tables describe the construction of the Rover robot, first the bumper, then the rear wheel, the eyes, the body and the antennas.

In the following tables, the numbers in parentheses are the length of the axes.

Warning : the yellow elastic of the bumper is not represented ; The connectors' wires are not represented ; the real antennas are not exactly the same as the ones on the pictures.

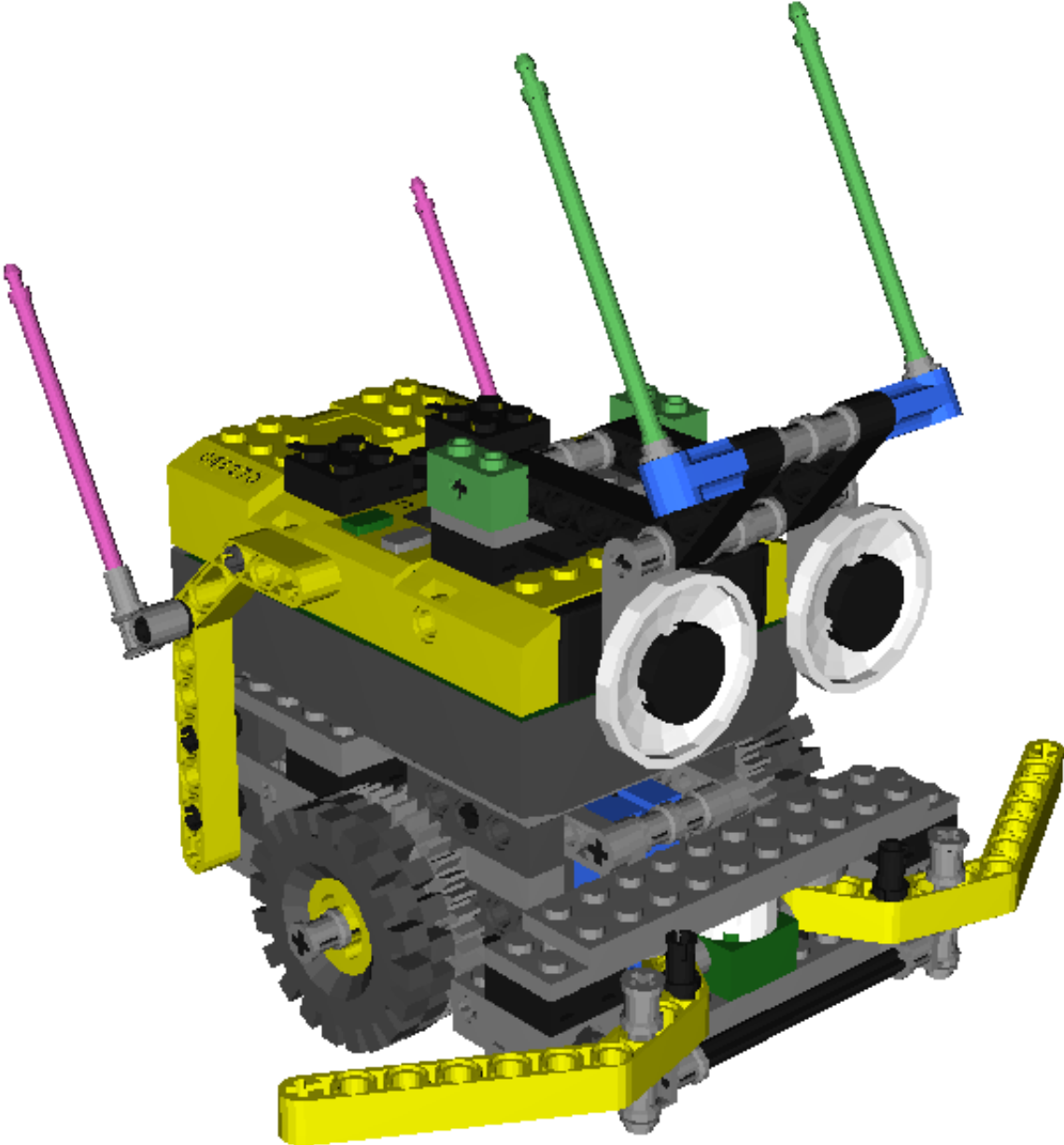
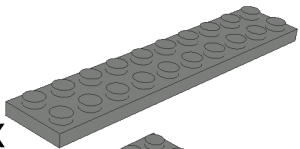
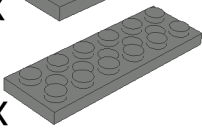
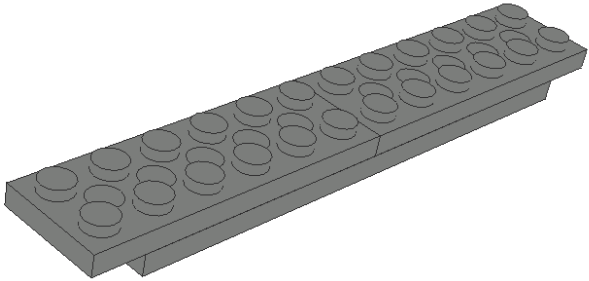
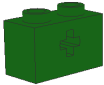


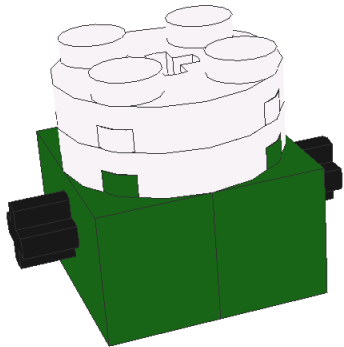


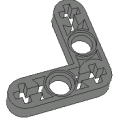
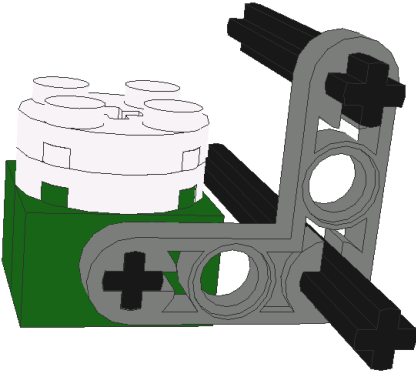


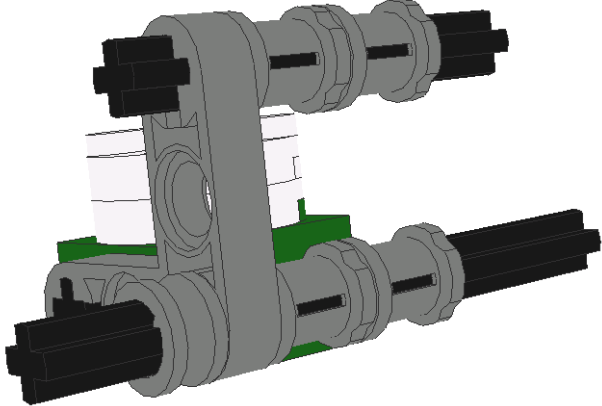

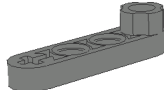

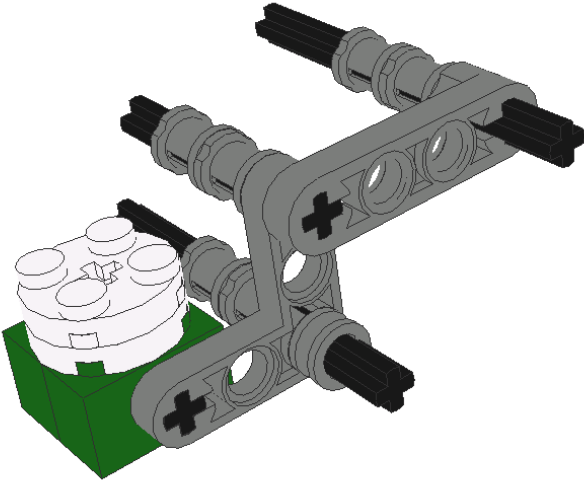


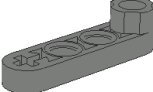
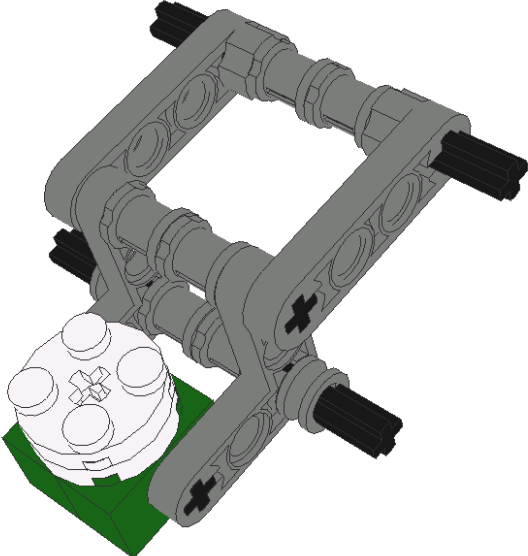
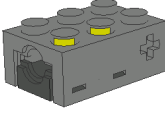
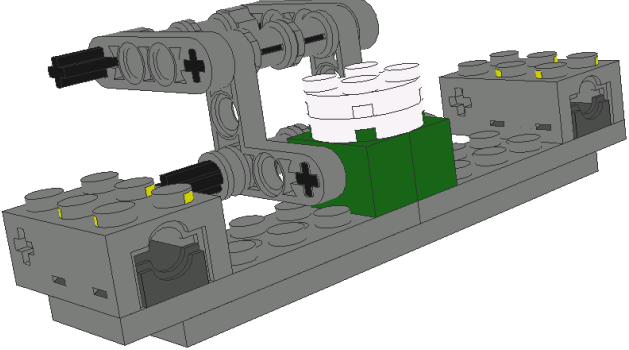



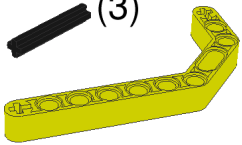
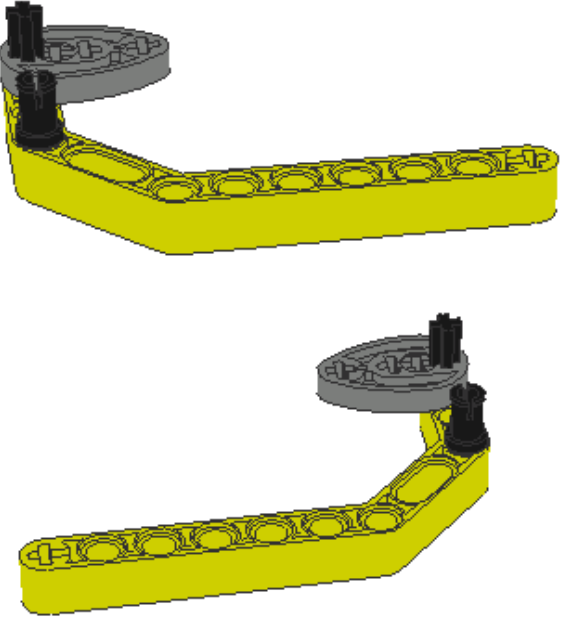
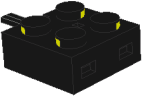
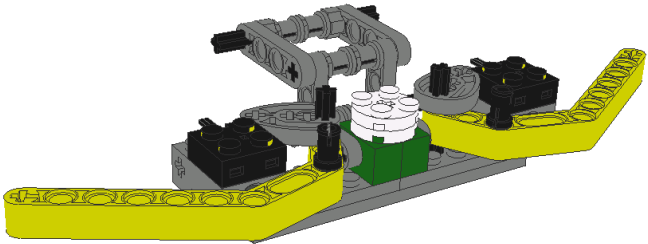
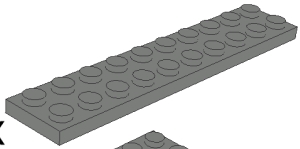
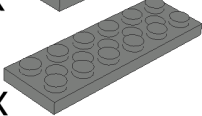
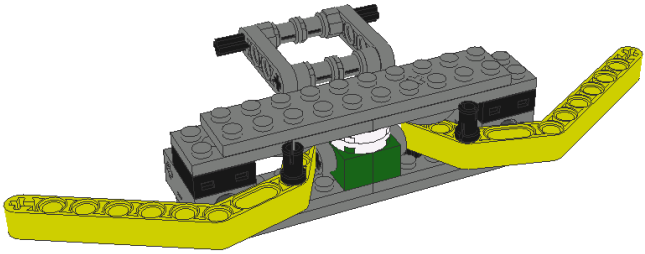

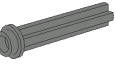



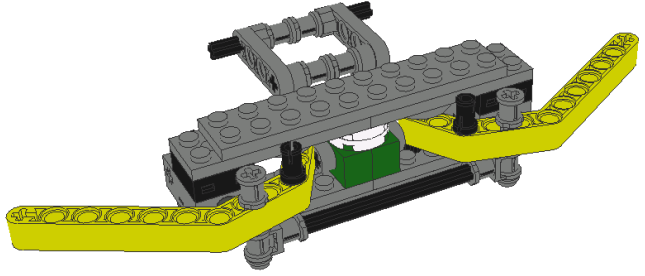







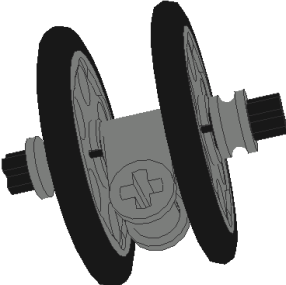



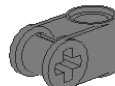
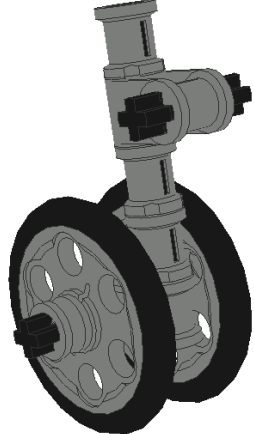

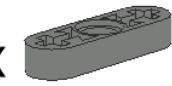


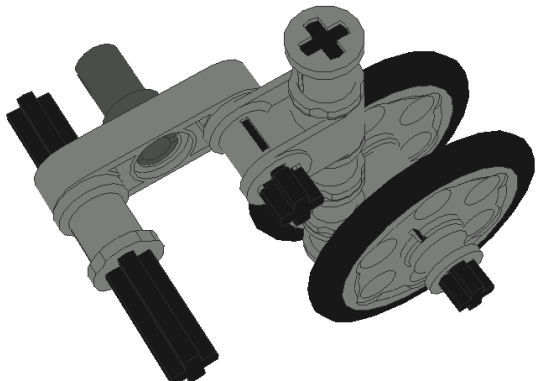
Figure 7.1: The Rover robot


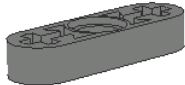
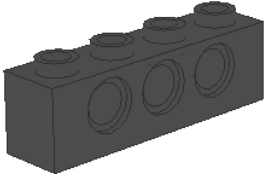
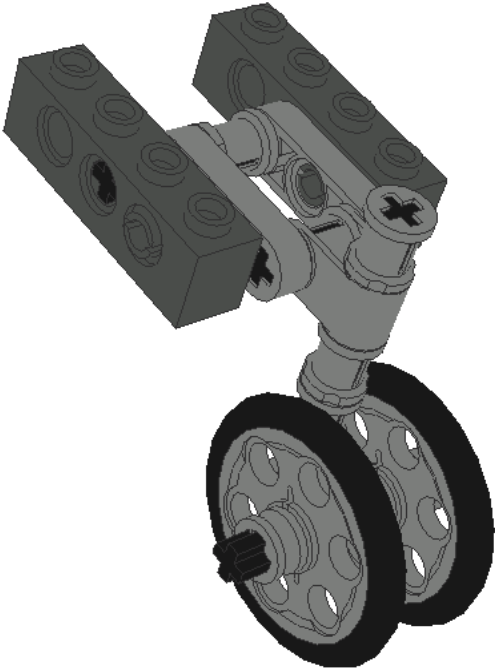
Step	Pieces	Modeling the bumper
1	<p>1x </p> <p>2x </p>	
2	<p>2x </p> <p>2x </p> <p>1x  (3)</p>	
3	<p>1x  (4)</p> <p>1x  (6)</p> <p>1x </p>	



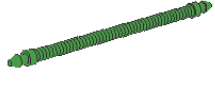



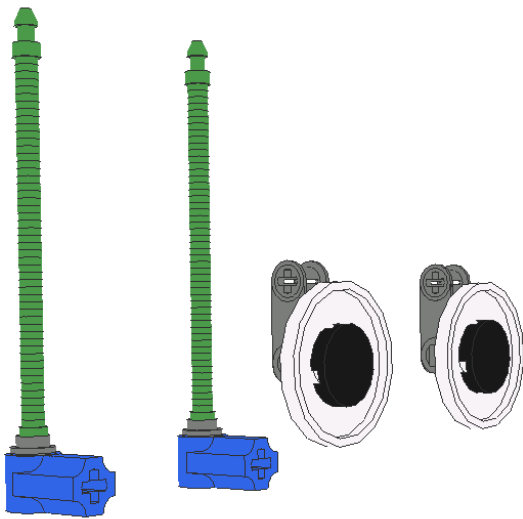
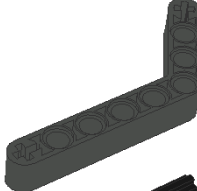


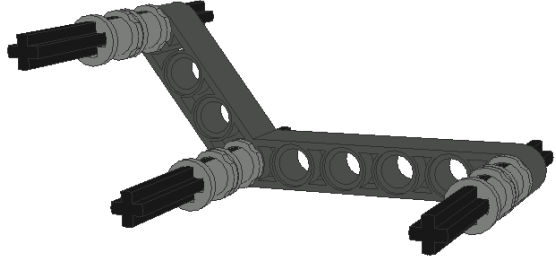
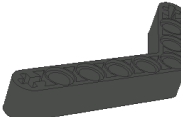

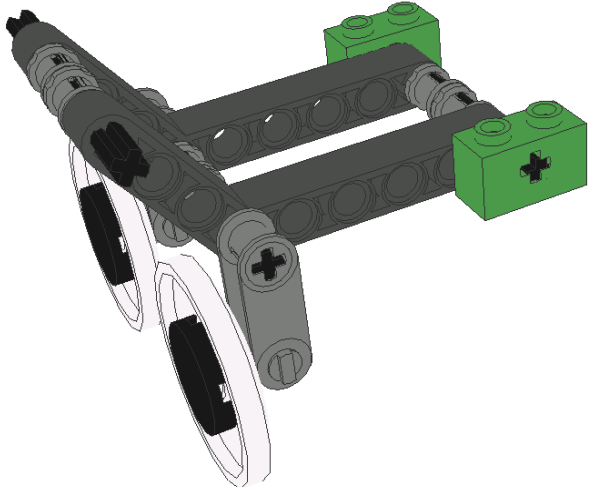
Step	Pieces	Modeling the bumper
4	1x  4x 	
5	1x  (6) 1x  2x 	
6	1x  1x  1x 	


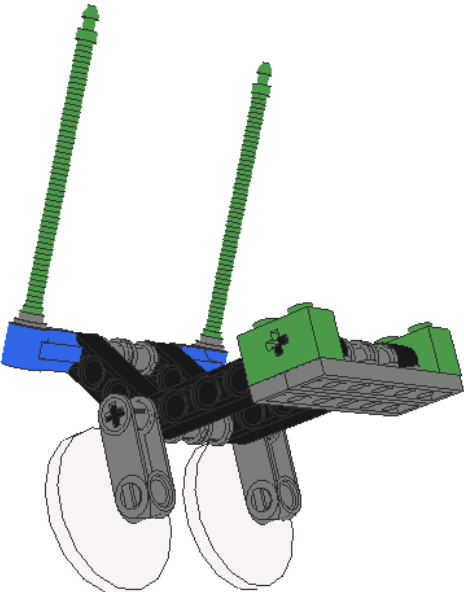
Step	Pieces	Modeling the bumper
7	<p>2x </p> <p>Step 1 + Step 6</p>	
8	<p>2x </p> <p>2x </p> <p>2x  (3)</p> <p>2x </p>	
9	<p>2x </p> <p>Step 7 + Step 8</p>	

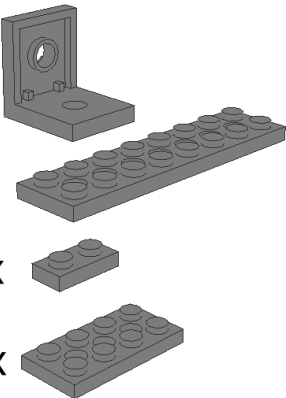
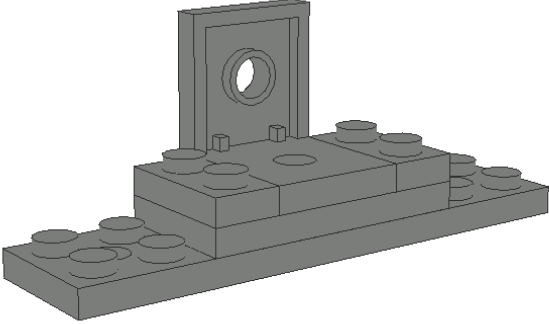
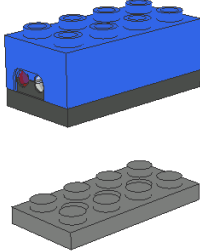
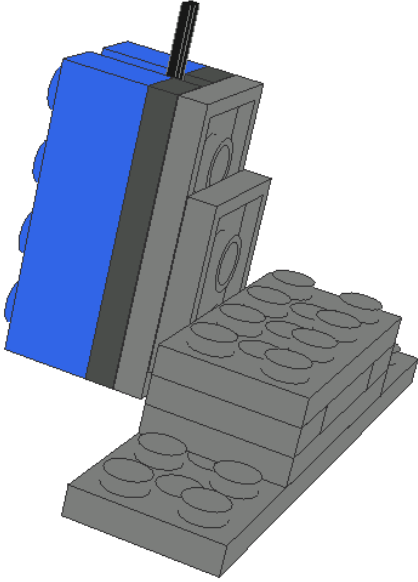
Step	Pieces	Modeling the bumper
10	<p>1x </p> <p>2x </p>	
11	<p>1x  (8)</p> <p>2x </p> <p>2x </p>	
12	<p>2x </p> <p>Step 10 + Step 11</p>	

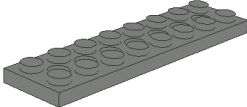


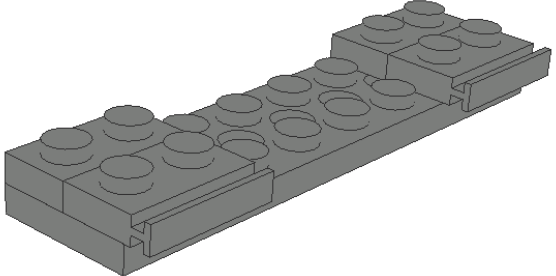
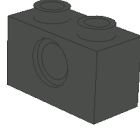
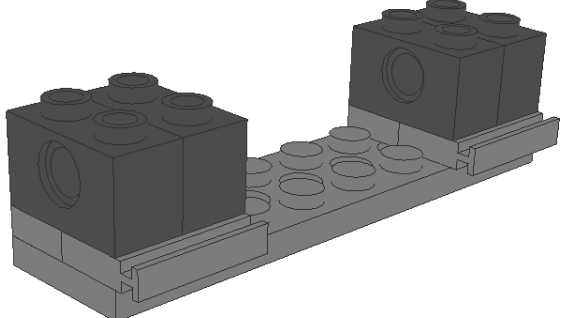

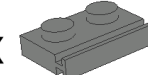
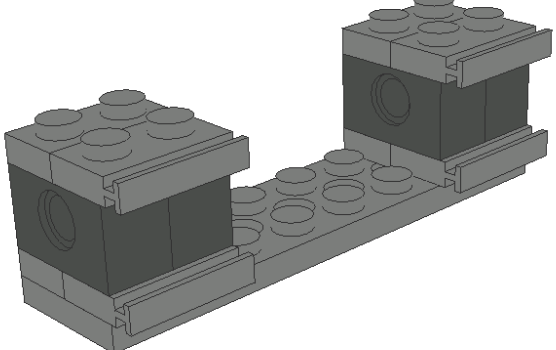


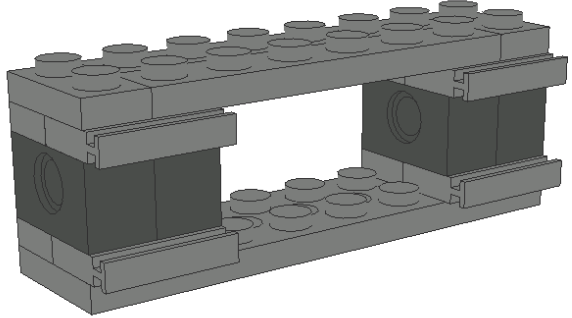
Step	Pieces	Modeling the rear wheel
1	2x  1x  1x  (4) 2x  2x 	
2	3x  1x  (5) 1x  (2) 1x 	
3	1x  1x  1x  (4) 1x 	

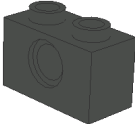
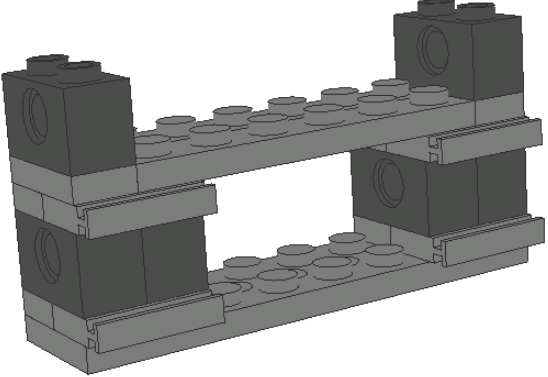
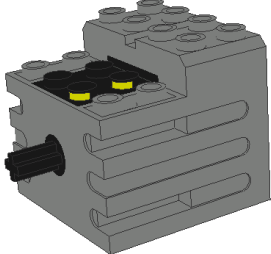
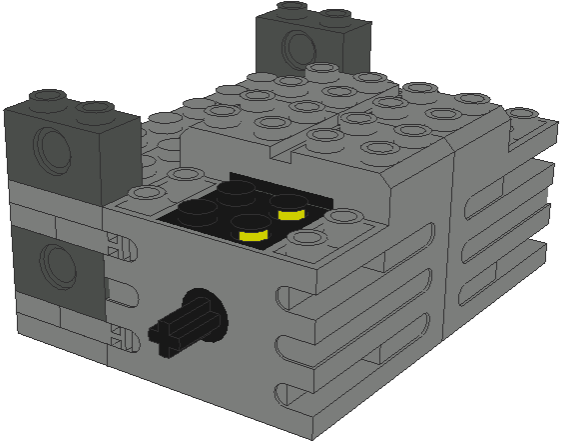
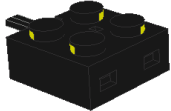
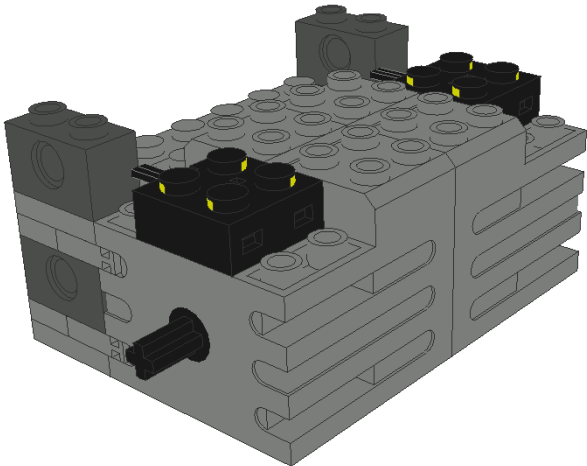
Step	Pieces	Modeling the rear wheel
4	<p>1x </p> <p>1x </p> <p>2x </p>	

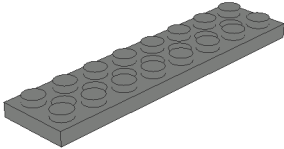
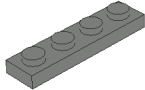
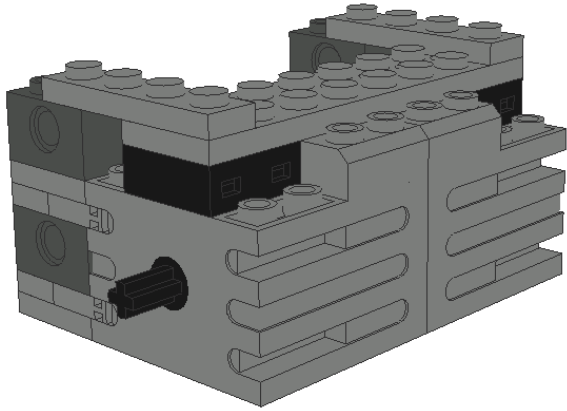

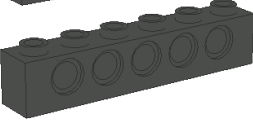
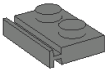

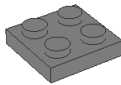
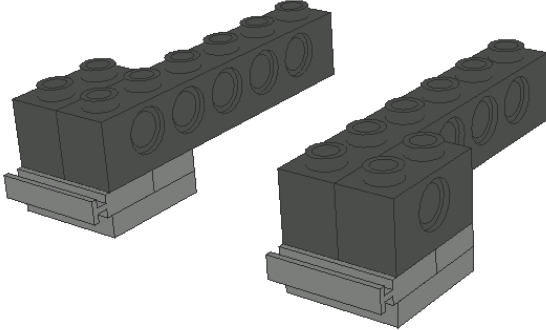



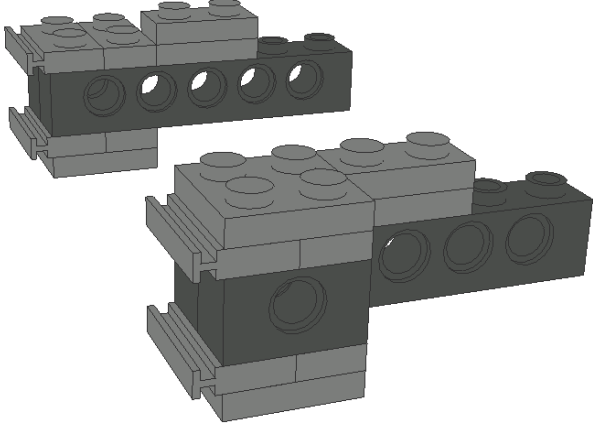
Step	Pieces	Modeling the eyes
1	<p>2x </p> <p>2x </p> <p>2x </p> <p>2x </p> <p>2x </p> <p>2x </p>	
2	<p>1x </p> <p>3x  (6)</p> <p>6x </p>	
3	<p>1x </p> <p>2x </p> <p>+ Step 1</p>	

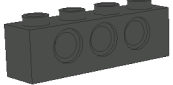

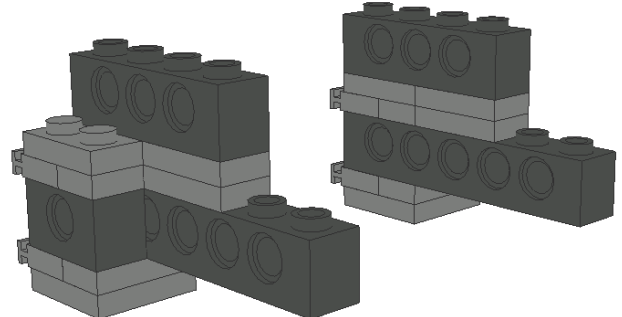
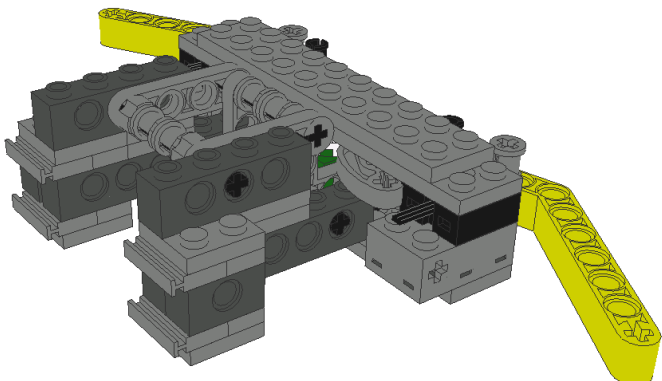
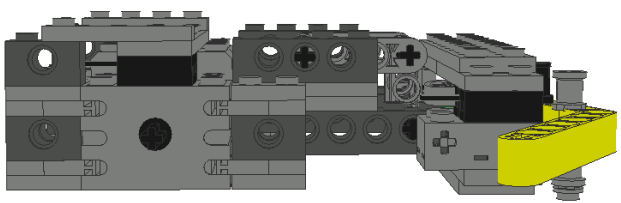
Step	Pieces	Modeling the eyes
4	2x 	

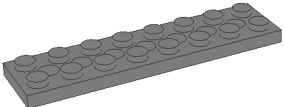
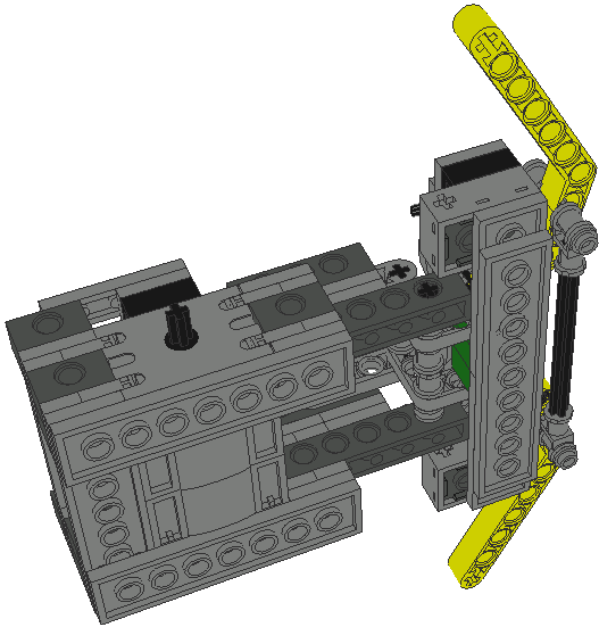

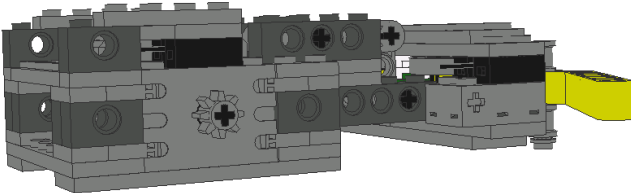

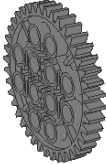




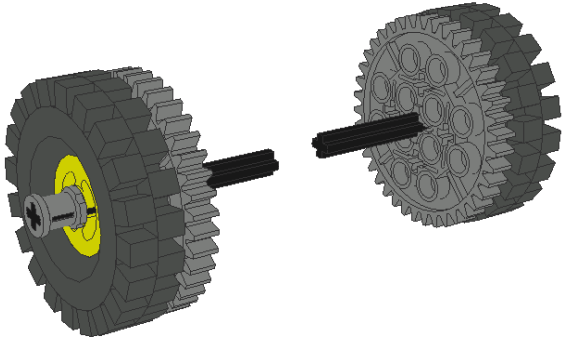
Step	Pieces	Modeling the light sensor
1	<p data-bbox="326 663 375 699">1x</p>  <p data-bbox="326 751 375 787">1x</p> <p data-bbox="326 829 375 865">2x</p> <p data-bbox="326 917 375 953">1x</p>	
2	<p data-bbox="326 1472 375 1507">1x</p>  <p data-bbox="326 1591 375 1627">2x</p>	

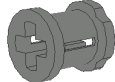
Step	Pieces	Modeling the body
1	1x  2x  2x 	
2	4x 	
3	2x  2x 	
4	2x  2x 	

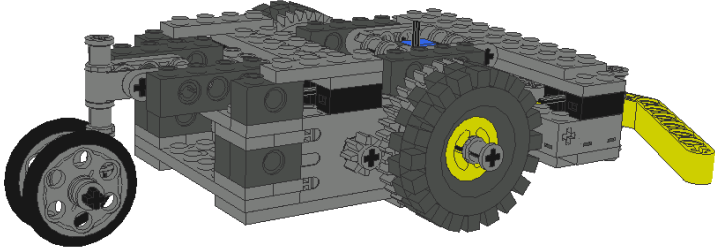
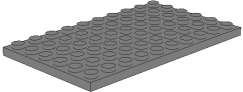
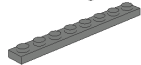
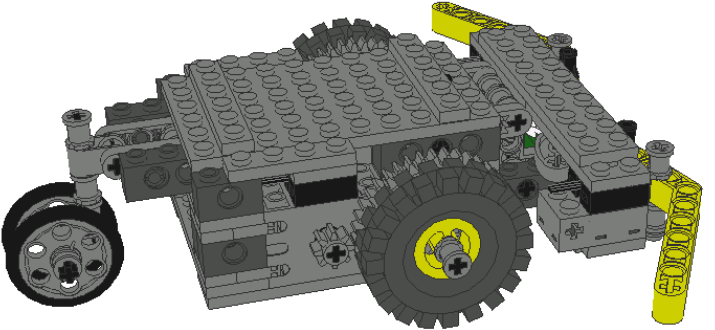
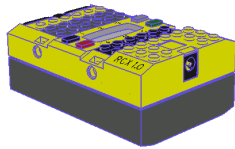
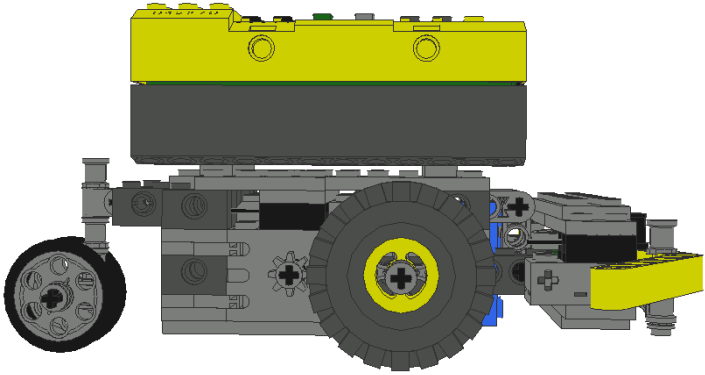
Step	Pieces	Modeling the body
5	<p>2x</p> 	
6	<p>2x</p> 	
7	<p>2x</p> 	

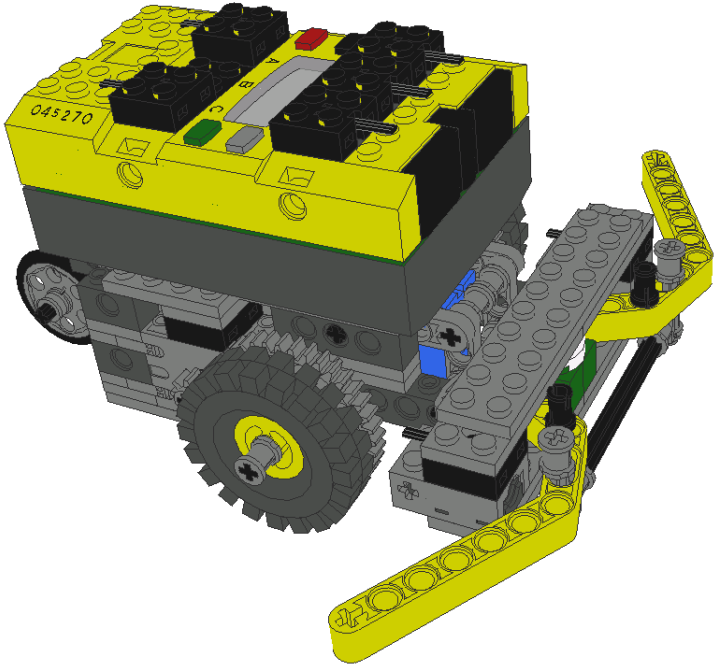
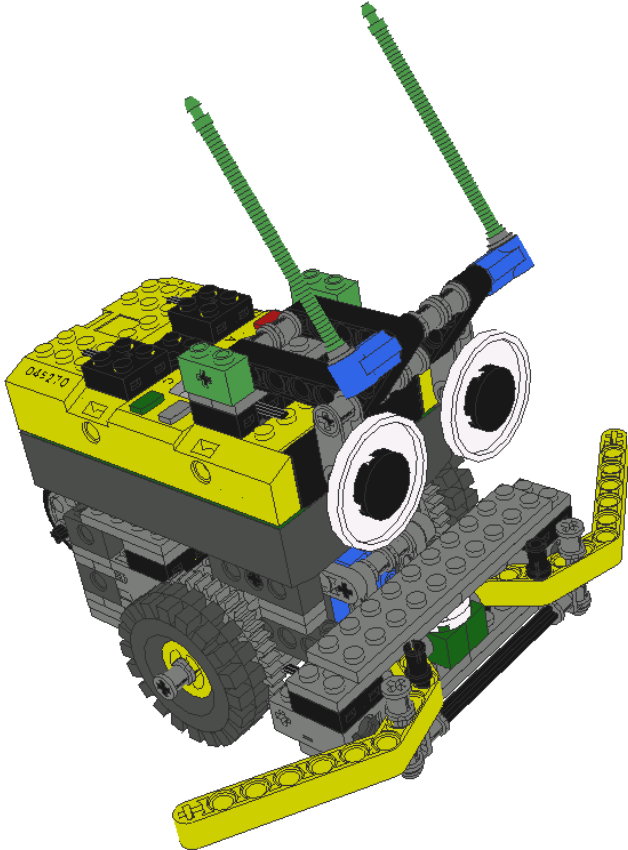
Step	Pieces	Modeling the body
8	<p>1x </p> <p>2x </p>	
9	<p>2x </p> <p>2x </p> <p>2x </p> <p>2x </p> <p>2x </p>	
10	<p>6x </p> <p>2x </p> <p>1x </p>	






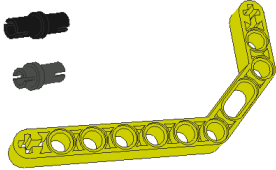
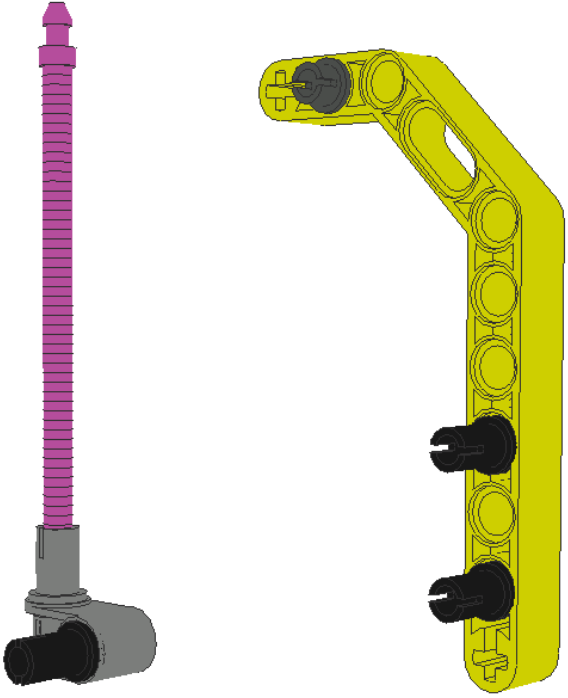
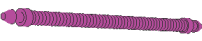




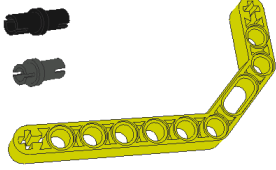

Step	Pieces	Modeling the body
11	2x  1x 	
12	Step 11 + Bumper	
13	Step 12 + Step 8	

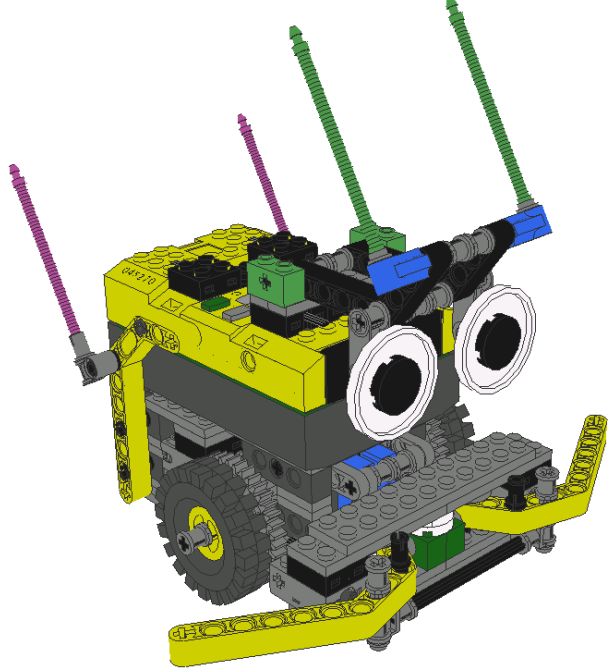
Step	Pieces	Modeling the body
14	2x 	
15	2x 	
16	2x  2x  2x  2x  (6) 2x  2x 	

Step	Pieces	Modeling the body
17	2x 	
18	Step 17 + Light Sensor	

Step	Pieces	Modeling the body
19	Step 18 + Rear Wheel	
20	1x  2x 	
21	1x 	

Step	Pieces	Modeling the body
22	<p>Connectors with wire :</p> <ul style="list-style-type: none"> 1- Left bumper 2- Light sensor 3- Right bumper A- Left motor B- Right motor 	 <p>A perspective view of a LEGO Mindstorms robot body. The top is a yellow Technic brick with two black motor connectors labeled 'A' and 'B'. A grey Technic beam is attached to the front, with a light sensor and two bumper sensors. The robot has two large grey wheels on the left side and a smaller grey wheel on the right. A yellow Technic beam is attached to the bottom right.</p>
23	Eyes	 <p>A perspective view of the same LEGO Mindstorms robot body as in step 22. Two green Technic axles are inserted into the front of the robot, passing through a blue Technic connector. Two white circular pieces are attached to the ends of the axles, representing the robot's eyes.</p>

Step	Pieces	Modeling the antennas
1	<ul style="list-style-type: none"> 1x  1x  1x  3x  1x  1x  	
2	<ul style="list-style-type: none"> 1x  1x  1x  3x  1x  1x  	

Step	Pieces	Modeling the antennas
3	Step 2 + Body	 A LEGO Mindstorms robot is shown from a three-quarter perspective. The robot has a yellow top deck and a grey base. It features two large black wheels on the left side and two smaller grey wheels on the right. The robot is equipped with four antennas: two purple and two green. The purple antennas are attached to the top deck, and the green antennas are attached to the front of the robot. The robot is also equipped with a blue motor and a yellow beam.

7.2 Webots model of the Rover robot

Webots already includes a model for the Rover robot you just built. So, you won't have to rebuild a virtual copy of this robot. The world file containing this model is named `rover.wbt` and depicted in figure 7.2. This file lies in the Webots `worlds` directory.

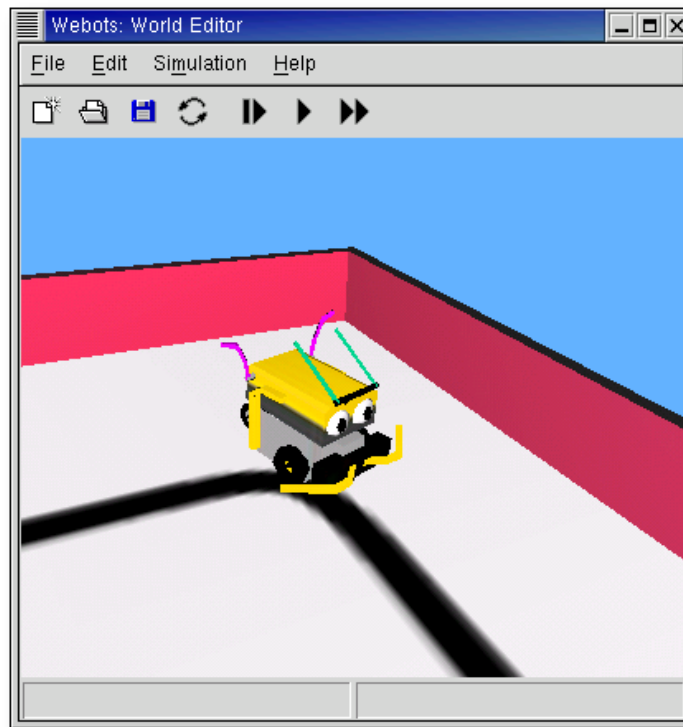


Figure 7.2: The Rover model in Webots

Before opening this file in Webots, Windows and Linux users should check that have properly installed java on their computer. The `java -version` command should answer this question.

Once you have launched Webots and opened the `rover.wbt` world, press the stop button to stop the simulation and study carefully the scene. Open the scene tree window by double-clicking on the robot. The scene is very simple. It contains a surrounding wall, a textured ground displaying a track and a Rover robot. Let's open the `DifferentialWheels` node corresponding to the Rover robot. Looking at its children list will reveal the robot is equipped with one distance sensor (looking down) and a couple of touch sensors, i.e., the bumpers. The two wheels are implemented as `Solid` nodes with "left wheel" and "right wheel" as names to allow the simulator to make them rotate when necessary. Finally the controller field of this `DifferentialWheels` node is set to "Rover". The fact the name of the controller begin with a capital letter means that the robot is programmed using the Java language. If you press the run button, the Rover robot will start moving on, following the track drawn on the floor, as programmed in its controller.

Let's have a look at the Java controller for the Rover robot. This controller lies in the `Rover` subdirectory of the `Webots controllers` directory. It contains a single Java source file named `Rover.java` and a `Makefile` file which are used for the compilation. To compile your controller, just type `make` in the `Rover` directory and it will produce a `Rover.class` java binary file that is used by `Webots` to control the robot.

Now, have a look at the source code. Open the `Rover.java` in your favorite text editor and try to understand what it contains. Useful comments should help you understand some details. If you are familiar with Java you will very easily understand everything since it is a very simple example. Basically, it gets references to the distance sensor and the touch sensors, enable these sensors for measurements each 64 milliseconds and enter an endless loop in which it perform a simple line following algorithm using only the distance sensor looking down to read the color of the floor. You may modify this program, recompile it and see how your modified version performs.

7.3 Transferring to the real Rover robot

7.3.1 leJOS

Now that you have a simulation model running as you like, it is time to transfer to the real robot to see if it behaves the same. In order to proceed, you will need to install the leJOS software. The leJOS software is a replacement firmware for the LEGO™ Mindstorms™ RCX brick. It is a Java Virtual Machine (JVM) that fits into the 32KB memory on the RCX hence allowing you to program your RCX in Java. The leJOS software is included on the `Webots` CD-ROM. Windows users will find a Windows version named `lejos.win32.2.1.0.zip` in the `devel` subdirectory of the `windows` directory. Macintosh and Linux users will find a source version named `lejos.2.1.0.tar.gz` in the `devel` subdirectory of the `common` directory. The documentation, including installation instructions, is located in the `common doc robots rcx` directory. Please take some time to read this documentation to understand how leJOS works. leJOS is also available from the leJOS web site¹.

7.3.2 Installation

Once you installed leJOS, as described in the installation instructions, you will have to upload the leJOS firmware into the RCX brick, replacing the LEGO™ operating system. Please follow the leJOS instructions to perform this installation. Note that you can easily revert to the LEGO™ operating system using the LEGO™ CD-ROM. Finally, you will have to set the `LEJOS_HOME` environment variable to point to the location where leJOS was installed. It is also necessary to add the leJOS `bin` directory into your `PATH` environment variable, so that you can use the leJOS tools from the command line.

¹<http://www.lejos.org>

7.3.3 Cross-compilation and upload

If everything was installed properly, cross-compilation and upload should be an easy task. Be sure that your robot is ready to receive a leJOS program. Go to the `Rover` controller directory and simply type `make -f Makefile.lejos` to launch the cross-compilation and upload processes. Note that it may be necessary to perform a `make clean` just before to remove any `class` file used for simulation. The cross-compilation process uses a different `class` file. Upload should happen just after cross-compilation and you should be able to run your controller on the real Rover robot.

7.3.4 How does it work ?

The `Makefile.kros` links your controller with a special Java wrapper class named `Controller`. This class lies in the `Webots lib` directory, in the `RCXController.jar` archive. It is a simple wrapper class between Webots Java API and leJOS API. Thanks to this system, the same Java source code can be used for both simulated robots and real robots. However, you should read carefully the limitations of leJOS Java implementation to avoid using Java features or libraries that are not supported by leJOS.

Chapter 8

Using the AiboTM robots

The goal of this chapter is to explain how to use Webots in connection with the Aibo ERS-series robots. Aibo is a four-legged dog-like robot developed by Sony Corp (www.aibo.com). While it is primarily intended for use as a toy, its flexibility in design and the ability to program on-board software using a C++ API (called OPEN-R) makes the Aibo a particularly interesting object for robotic research as well. In particular, the Robocup Sony Four-Legged Robot League (www.openr.org/robocup) is very popular among roboticists throughout the world.

The following Aibo models are currently supported in Webots: ERS-210, ERS-7.

8.1 Introduction

We will assume that you already have a basic knowledge of the Webots environment and know how to perform simple tasks like opening a file and navigating through the world contained therein (if you do not, please refer to chapter 2). For additional Aibo documentation (in particular, the OPEN-R SDK guides to which we sometimes make reference in this chapter), please visit the official OPEN-R website (openr.aibo.com).

The following conventions are used: *WLAN* stands for "wireless local area network"; *Control Panel* is the Aibo ERS-series control applet integrated into Webots; *RCServer* stands for "remote control server", it is the on-board software running on Aibo to be used in conjunction with the Control Panel to remotely control your Aibo; *Memory stick* is your Aibo Programming Memory Stick supplied with the robot; finally, *ersxxx* always refers to your Aibo ERS-XXX model.

The following relevant resources are located under Webots installation directory:

- `controllers/ers*/`: sample controller source directories for the Aibo ERS-series robots (see also section 8.6);



Figure 8.1: Supported Aibo ERS-7 (left) and ERS-210 (right) robots

- `data/mtn/ersxxx/`: MTN motion sequence data files for the Aibo ERS-XXX model (see also subsection 8.5.3);
- `transfer/openr/`: transfer-related data (software to copy to the Memory stick, Aibo definition includes for your controllers, etc.);
- `worlds/aibo_ers*.wbt`: world definition files containing at least one Aibo robot (see also section 8.3);
- `objects/aibo_ers*.wbt`: Webots object files containing a single Aibo model, for imports (see also section 8.4).

Note: You should copy the relevant files to the corresponding paths in your user directory for ease of use. Typically, you will want to copy the Aibo worlds and controller files, as well as the MTN motion sequence data files.

8.2 Hardware configuration

Your Aibo can be connected to the outside world by means of a wireless 802.11b network connection. We shall first describe the necessary networking setup for both your Aibo and your PC. We shall then explain how to install our custom on-board Aibo software (called `RCServer`) which is required to make full use of the Aibo Control Panel integrated into Webots. The actual communications with the robot are carried out via TCP/IP.

8.2.1 Hardware requirements

You will need obviously an Aibo robot. For now, only the ERS-210, ERS7 and ERS7M2 are supported. If you have an ERS-210 Aibo, you will also need the 802.11b wireless card which

comes as an option with the robot. It has to be installed in the body of the robot, according to the provided setup instructions.

8.2.2 Setting up a wireless link with Aibo

The configuration of the Aibo robot to be part of a wireless network is described in detail in the *Aibo User's Guide (PC Network)*. There are several ways to do that; we shall explain here how to setup a peer-to-peer (ad hoc) connection between a PC with WLAN capabilities and the Aibo. If you are running an infrastructure-based WLAN network (i.e., you connect to an access point on your network), please refer to the relevant sections of your Aibo documentation. In particular, section 3.2.2 "How to set up WLANCONF.TXT" of the *OPEN-R SDK Installation Guide* can prove helpful.

Configuring Aibo in ad-hoc mode

Aibo's network configuration is done by means of a configuration file (WLANCONF.TXT) placed onto the Memory stick at location /MS/OPEN-R/SYSTEM/CONF/. To set up the default (ad hoc) configuration, simply copy the WLANDFLT.TXT file found at same location on your Memory stick to WLANCONF.TXT. You're done!

Important: Do *not* edit the WLANDFLT.TXT file! Always make changes to a copy saved as WLANCONF.TXT, as explained above.

Configuring your PC in ad-hoc mode

To establish a peer-to-peer connection with Aibo, your PC will need to have an 802.11b-compatible network adapter installed, and you will need to setup your wireless link as follows:

1. Use the configurator supplied with your WLAN (802.11b) adapter to create a new "Ad-hoc" or "Peer-to-peer" profile, called for example "Aibo".
2. Edit your new "Aibo" profile settings and set the following parameters:

```
ESSID    = AIBONET (network name)
CHANNEL  = 3
WEP      = WEP64 (40-bit key encryption)
WEPKEY   = AIBO2 (ascii) or 0x414924f32 (hex)
IP       = 10.0.1.1
MASK     = 255.255.255.0
```

For help on how to set up above parameters, please refer to your operating system and/or wireless adapter documentation.

Note: If you have changed the default settings for your Aibo network configuration in the WLANCONF.TXT file on the Memory stick, you must change the above parameters accordingly.

8.2.3 Mounting the Memory stick

Before setting up the Memory stick for use with Webots, you will need to plug and mount it on your computer. If you already know how to do this, you may safely skip this subsection.

Using the Memory stick with Windows

Your Memory stick reader comes with a Windows drivers CD which you will need to install in order to be able to use it with your Windows system. Please refer to the CD installation process for details. Once you installed the drivers, plug the card reader in and insert the Memory stick, the system should see and mount it automatically.

Using the Memory stick with Mac OS X

Mac OS X already has all required drivers, so all you need to do is plug the card reader in and insert the Memory stick, the system should see and mount it automatically.

Using the Memory stick with Linux

The following procedure explains how to mount and use the Memory stick on a Linux-based system.

1. Plug your Memory stick reader into a USB port on your computer. Most Linux distributions support hotplug technology for USB devices, however if your computer does not see the reader, try plugging it in before turning on the power.
2. After you log into a shell, make sure that the USB reader is recognized by the system. You can do so by typing:

```
$ cat /proc/bus/usb/devices | grep S:
```

and you should see something like this:

```
S: Product=USB OHCI Root Hub
S: SerialNumber=ccacb000
S: Manufacturer=SCM Microsystems Inc.
S: Product=eUSB MemoryStick Reader
S: SerialNumber=0000000011CF
```

3. Create an empty directory to mount your USB reader to e.g., /mnt/usb:

```
$ mkdir /mnt/usb
```

4. Finally, mount the Memory stick (you must have supervisor privileges):

```
$ mount /dev/sda1 /mnt/usb
```

After mounting the Memory stick to the specified directory (`/mnt/usb`), you can use it just like any other removable storage device.

Important: When you `'ls'` the Memory stick for the first time, you will find a file named `memstick.ind`. Do *not* delete it, since the Memory stick is to be used for your Aibo robot.

8.2.4 Setting up your Memory stick

In order to benefit from the Webots remote control capabilities with Aibo, we need to set up the remote control (`RCServer`) software running on Aibo. This is done by copying some data onto Aibo's Memory stick.

A complete set of files to be copied onto your Memory stick is available in Webots *OPEN-R transfer directory*, located in your Webots installation directory as:

```
transfer/openr/ersXXX/
```

To setup your Memory stick, follow this simple procedure:

1. Plug and mount your Memory stick on your computer (see subsection 8.2.3).
2. Copy *all* of the content of the transfer directory onto the Memory stick.
3. Unmount the Memory stick and insert it into your Aibo. The appropriate slot is located on the belly of your robot, right behind the battery. You are ready to go!

Important: Always *unmount* the Memory stick before removing it from the reader! Under Linux, use `umount /mnt/usb`; under Mac OS X, drag your USB reader icon to the trash bin; under recent Windows editions, use the 'Unplug or Eject hardware' icon located on the taskbar.

When you power up your Aibo, the green power light blinks, then the robot stretches its legs and gets up: it is now ready to connect and process remote commands.

8.3 Running the simulation

Launch Webots: on Windows, double-click on the ladybug icon, on Linux, type `webots` in a terminal. Go to the **File/Open** menu item and open the file `worlds/aibo_ersXXX.wbt` (where `XXX` corresponds to your Aibo ERS-XXX model). That world contains an Aibo robot in a stadium-like arena.

Note: You can make the Aibo control panel (see section 8.5) appear by selecting the robot and choosing **Show robot window** in the **Simulation** menu, or simply by double-clicking the robot (if the robot window is already open, double-clicking the robot opens the **Scene tree** instead).

8.3.1 Default Aibo world

The default Aibo world comes with a pre-configured controller, which will make the robot walk for a few steps. When you start the simulation, you will see the Aibo model walk forward, and kick the red ball. Of course, you can also edit the world and select another controller (change the controller property of Aibo's root CustomRobot node), or get rid of it altogether (simply select void as a controller). Available Aibo controllers are located in Webots controllers/ers*/ directories (see section 8.6).

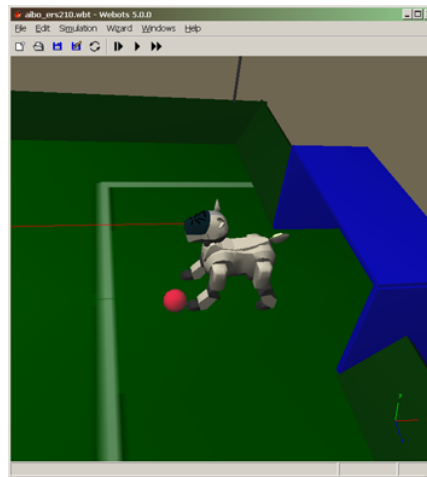


Figure 8.2: Default Aibo world (ERS-210)

8.3.2 Other Aibo worlds

There are other worlds available which contain an Aibo robot. We will briefly describe each of them and explain their uses.

Aibo rough world

This world is located in `worlds/aibo_ersXXX_rough.wbt`. It contains a single Aibo robot placed onto a rough terrain structure. Like in the default world, when you run the simulation the robot will walk forward for a few steps. You can see how the walk sequence is disturbed by the ground relief. This world is primarily designed to test walking algorithms on uneven terrains.

Aibo soccer world

This world is located in `worlds/aibo_ersXXX_soccer.wbt`. It contains two teams of 3 Aibo robots, each team bearing its own color. Again, when you run the simulation, all robots will start walking and stop after a few paces.

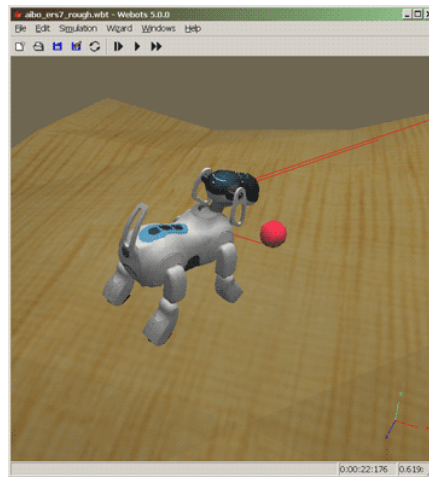


Figure 8.3: Aibo rough world (ERS-7)

Note: This particular simulation requires considerable computing power, because all 6 Aibo robots' default controllers access their cameras at each cycle.

This world is designed to prepare your robots for the Robocup Sony Four-Legged Robot League¹. Use it to test your algorithms in simulation before you transfer them to the real robots. This should spare your robots in case you use particularly stressful mechanisms like the now famous *knee-walk* algorithm.

Note: For this world, robot camera windows are hidden. To display a particular robot's camera, find the robot's head camera node in the **Scene Tree** and set its `display` property to `TRUE`.

The head camera node is located in the **Scene Tree** as

```
ERS210->NECK_TILT->HEAD_PAN->HEAD_ROLL->HEAD_CAM
```

node for the ERS-210, and as

```
ERS7->NECK_TILT->HEAD_PAN->HEAD_TILT->HEAD_CAM
```

node for the ERS-7 model.

Aibo models world

This world is located in `worlds/aibo_ersXXX_models.wbt` world file. It contains available Aibo model objects for given ERS series, placed onto a simple flat checked ground. Unlike the other worlds, the robots have no associated controller, so running the simulation will have no spectacular effect.

These worlds are designed to showcase available model objects, as well as for use with manual control (using the Control Panel, see section 8.5), e.g., to test new or existing MTN motion sequences, or establish a remote connection and observe the real robot.

¹<http://www.openr.org/robocup>

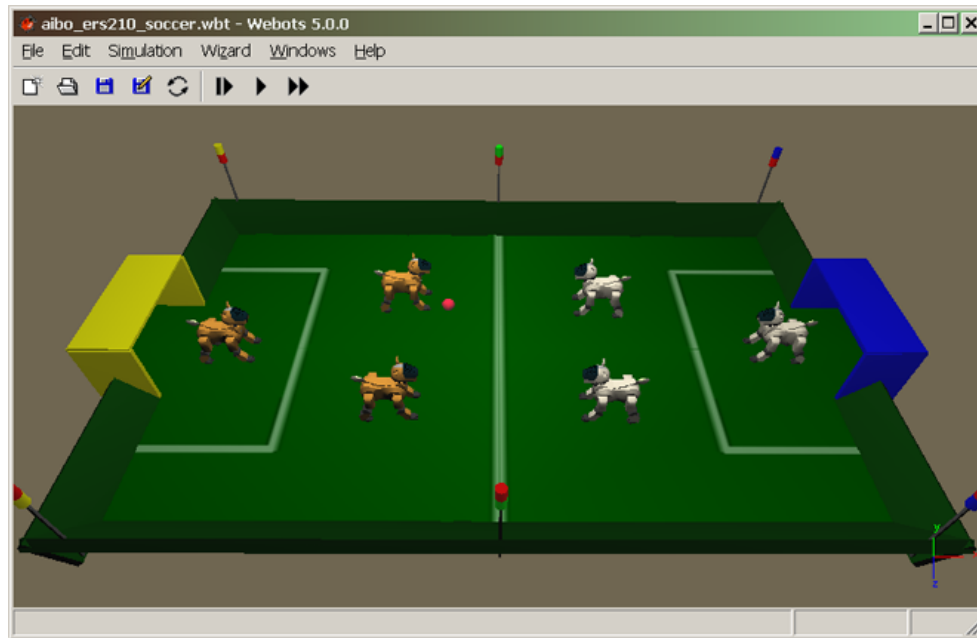


Figure 8.4: Aibo soccer world (ERS-210)

8.4 Understanding the Aibo models

Each of the above worlds contains at least one Aibo robot object, which may be found separately in Webots `objects/` subdirectory. In this section, we discuss the two models and explain their similarities and individual differences.

8.4.1 Reuseable Aibo objects

If you need to build your own custom world and include an Aibo robot, you may simply import one of the existing Aibo objects, shown in figure 8.5, into the **Scene Tree**.

Available ERS-210 objects are:

- `objects/aibo_ers210.wbt`: default (silver) appearance;
- `objects/aibo_ers210_bronze.wbt`: alternative (bronze) appearance.

Available ERS-7 objects are:

- `objects/aibo_ers7.wbt`: default (silver) appearance;
- `objects/aibo_ers7_blue.wbt`: alternative (blue) appearance.

Note: Object paths are indicated relative to Webots installation directory. Alternative appearance is used, e.g., for the soccer worlds, see section 8.3.

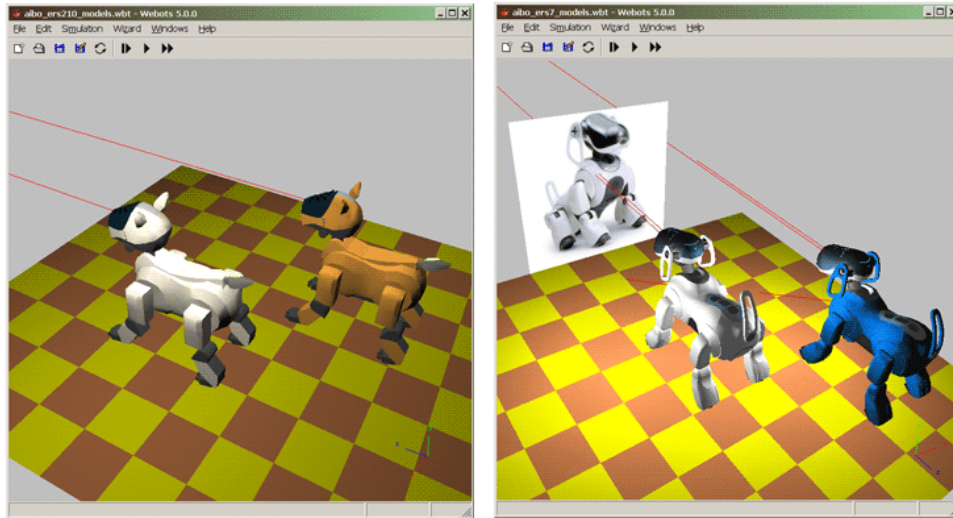


Figure 8.5: Aibo ERS-210 (left) and ERS-7 (right) models worlds

8.4.2 General model overview

The ERS-210 (figure 8.6) and ERS-7 (figure 8.8) models have a lot in common. On the outside, both are of course four-legged dog-like robots. From a 3D-modelling point of view, the Aibo robot is made up of the following parts:

- a head attached to a neck, with 3 degrees of freedom;
- a static body, to which all other parts are attached;
- four legs with 3 degrees of freedom each;
- finally, a tail with 2 degrees of freedom.

The legs have two shoulder joints (*swing* and *flap* movements) and one knee joint. There is a spherical touch sensor located under each paw to detect ground contact. Both models contain an accelerometer (which however has no simulated equivalent) and a few other sensors which we describe further on.

We also provide some model description text files, located in Webots `worlds` directory:

- `worlds/aibo_ersXXX.model.txt`: VRML structure with various parameter indications and primitive names-to-nodes correspondence for the ERS-XXX model;
- `worlds/aibo_ers7ers210_prm_map.txt`: contains mapping of primitives common to both ERS-series robot models.

Controllable devices (primitives) naming

As per Sony OPEN-R conventions, Aibo's controllable devices are called *primitives*. Each primitive is identified by a unique string of characters: its primitive name. Primitive names all start with the "PRM: " prefix. Accordingly, the Webots Aibo models contain nodes which are named after the corresponding real-life primitives (when an equivalent simulation node is available).

For ease of reading, we shall frequently address primitives by their *PIDs* instead of fully qualified primitive names. PID stands for "primitive ID" and refers to a constant useable for your programs (see section 8.6). Consider PIDs as unique *numeric* identifiers through which is it possible to obtain the real primitive name for each device. Individual model sections give the correspondence between PIDs and their associated primitive names.

Common primitives

In this section, we discuss *shared* primitives, which both supported Aibo models have (more or less) in common. For primitives specific to each model, please refer to the relevant subsection below.

Common primitives and their availability are listed in table 8.1. A mapping of common primitive names may also be found in `worlds/aibo_ers7ers210_prm_map.txt`. Please note that while corresponding primitives perform similar functions in both models, they still have individual differences, e.g., hardware angular limits for joints, physical positions, etc.

In addition to the listed primitives, both models have sensors which allow them to measure distances. The reason why those are not listed above is because the newer ERS-7 model was outfitted with 3 individually accessible distance sensors, while the older ERS-210 had only 1 distance sensor, or PSD (position sensing device).

As you can see, most of the common devices are implemented in our models. Let us see which are not, or require special treatment:

- *Pressure sensors* (located on the top of the head and top of the body of the robot): in this particular case, there is no apparent need for these sensors, which are intended to be used "for fun" to play with the robot, and are therefore useless in simulation;
- *Speaker/microphone*: because Webots simulation engine does not yet handle sounds, these cannot have an equivalent simulated device;
- *Accelerometer*: Webots does not yet have a type of node useable to emulate an accelerometer; when a relevant node becomes available, the simulated models may be outfitted with these too;
- *Plungers*: there is no special node in Webots for modelling plungers, i.e., joints which only have two positions (on/off); Aibo has two of these devices (one for each ear), which are listed as "servo (plunger)" in table 8.1; these are modelled as regular servos, with

the following convention: `max = on`, `min = off` (`min` and `max` being the minimum, resp. maximum, angular joint limit defined for the servo node).

We shall now consider each model separately and provide detailed information regarding model-specific primitives and their support in Webots.

8.4.3 Aibo ERS-210 model

This model's default object is located in `objects/aibo_ers210.wbt` file. Its VRML structure with various parameter indications and primitive names-to-nodes correspondence can be found in `worlds/aibo_ers210_model.txt`. ERS-210 model schematics are shown in figure 8.6; note that unlike the indicated measurements (provided by Sony), standard Webots units are [m] for position/distance and [rad] for angles.

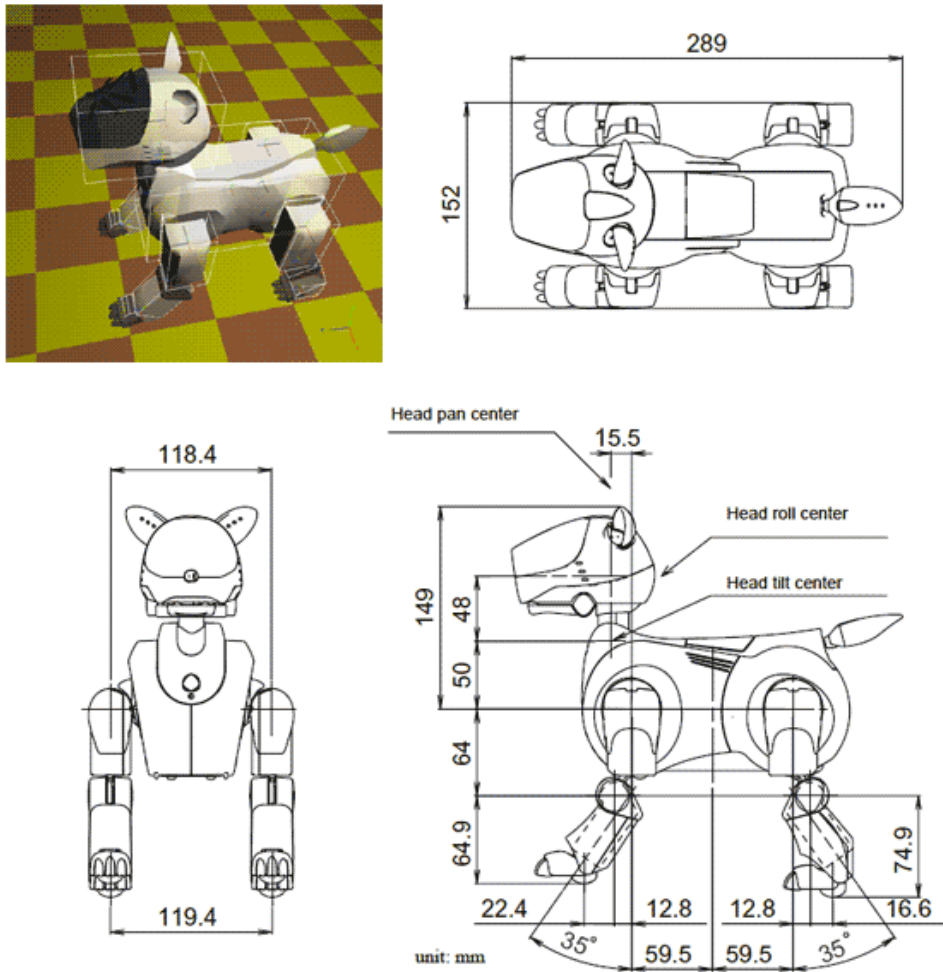


Figure 8.6: ERS-210 model overview (schematics provided by Sony)

Description	ERS-7 PID [1]	ERS-210 PID [2]	Node type
Neck tilt	HEAD_TILT1	HEAD_TILT	servo
Head pan	HEAD_PAN	HEAD_PAN	servo
Head tilt	HEAD_TILT2	- (see <i>Head roll</i>)	servo
Head roll	- (see <i>Head tilt</i>)	HEAD_ROLL	servo
Mouth (jaw)	HEAD_MOUTH	HEAD_MOUTH	servo
Left ear	HEAD_LEFT_EAR	HEAD_LEFT_EAR	servo (plunger)
Right ear	HEAD_RIGHT_EAR	HEAD_RIGHT_EAR	servo (plunger)
Chin touch sensor	HEAD_CHIN	HEAD_CHIN_SWITCH	-
Color camera	HEAD_CAMERA	HEAD_CAMERA	camera
Microphone	HEAD_MIC	HEAD_MIC	-
Speaker	SPEAKER	HEAD_SPEAKER	-
Left front leg, shoulder swing	LFLEG_J1	LFLEG_J1	servo
Left front leg, shoulder flap	LFLEG_J2	LFLEG_J2	servo
Left front leg, knee	LFLEG_J3	LFLEG_J3	servo
Left front leg, paw sensor	LFLEG_PAW	LFLEG_PAW	touch sensor
Left rear leg, shoulder swing	LRLEG_J1	LRLEG_J1	servo
Left rear leg, shoulder flap	LRLEG_J2	LRLEG_J2	servo
Left rear leg, knee	LRLEG_J3	LRLEG_J3	servo
Left rear leg, paw sensor	LRLEG_PAW	LRLEG_PAW	touch sensor
Right front leg, shoulder swing	RFLEG_J1	RFLEG_J1	servo
Right front leg, shoulder flap	RFLEG_J2	RFLEG_J2	servo
Right front leg, knee	RFLEG_J3	RFLEG_J3	servo
Right front leg, paw sensor	RFLEG_PAW	RFLEG_PAW	touch sensor
Right rear leg, shoulder swing	RRLEG_J1	RRLEG_J1	servo
Right rear leg, shoulder flap	RRLEG_J2	RRLEG_J2	servo
Right rear leg, knee	RRLEG_J3	RRLEG_J3	servo
Right rear leg, paw sensor	RRLEG_PAW	RRLEG_PAW	touch sensor
Tail tilt	TAIL_TILT	TAIL_TILT	servo
Tail pan	TAIL_PAN	TAIL_PAN	servo
Accelerometer (front-back)	ACCEL_Y	ACCEL_Y	-
Accelerometer (right-left)	ACCEL_X	ACCEL_X	-
Accelerometer (up-down)	ACCEL_Z	ACCEL_Z	-

Table 8.1: Common available ERS-7/ERS-210 primitives correspondence; PIDs are enumerated constants useable to access a primitive name from your program, they are located in a header file under `transfer/openr/include/`: [1] see `ers7.h`, [2] see `ers210.h`

ERS-210 primitive IDs are defined in `transfer/openr/include/ers210.h`. Correspondence between common PIDs and their primitive names is listed in table 8.2. Devices specific to the ERS-210 model and their availability are given in table 8.3.

Note: For physical details, feedback devices output range, LED positions, etc. please refer to OPEN-R SDK online documentation², "*Model Information for ERS-210*".

As you can see, most of ERS-210 devices are implemented in Webots. Let us see which of the ERS-210 *specific* devices are not:

- *Pressure sensors*: like the common pressure sensors, these are useless in simulation and therefore not modelled;
- *Thermo sensor*: because Webots simulation engine does not handle temperature, there is no equivalent for a thermo sensor device.

(For an explanation of common unsupported devices, please refer to subsection 8.4.2.)

Ear plungers

As mentioned previously, plungers are modelled as regular servos whose `min` represents the *off* state, and `max` represents *on*. The ERS-210 has two plungers: the ears. It is of course possible for a controller to command them to any angle between `min` and `max`, however only the extreme positions are meaningful as far as the real robot is concerned.

ERS-210 ears rotate around a 45 degree axis looking upward. By default, ears are "pricked", i.e., are in *on* state. Switching them *off* makes them look like they lie flat, i.e., giving the dog a "humble" look.

Distance sensor (PSD)

The distance sensor or PSD (*Position Sensing Device*) is located in the head of the robot, looking straight ahead, and has a documented range of 90 cm, with a minimum of 10 cm. The response curve of the model sensor, which has been established from experimental measurements carried out on the real robot, is shown on figure 8.7. Its lookup table is as follows:

```
lookupTable [ 0    100  0
              0.1  100  0
              0.3  300  0
              0.4  400  0.01
              0.5  480  0.01
              0.6  560  0.01
              0.7  630  0.02
```

²<http://openr.aibo.com>

0.8	680	0.03
0.9	740	0.05
1	790	0.08
1.1	840	0.08
1.2	870	0.08
1.5	870	0.08
1.8	700	0.08
2.1	870	0.08
2.3	900	0

]

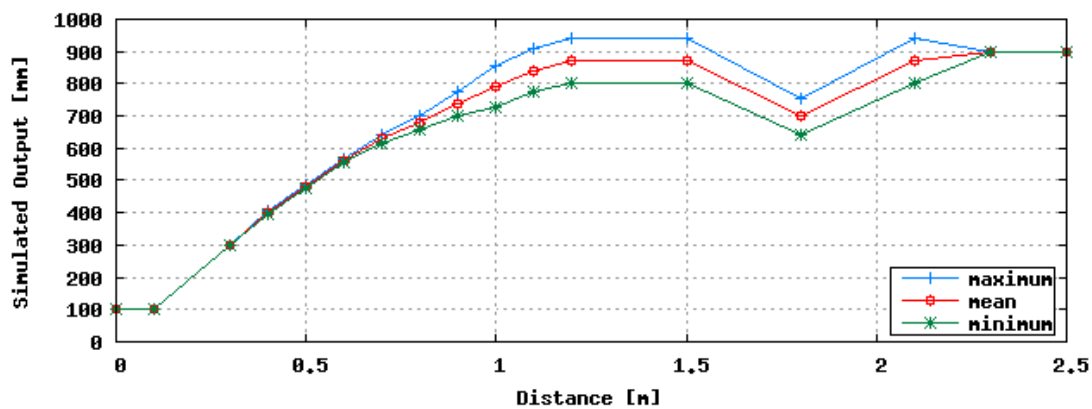


Figure 8.7: Response curve of the ERS-210 position sensing device

Tail LEDs

Most LED lights of the ERS-210 model are fully functional: face lights and mode indicator light have proper equivalent devices in simulated model. Tail LEDs, however, are only "virtually" supported, i.e., the primitive is accessible like any regular LED device but there will be no *visible* change to the simulated robot (the internal state of the LED is still set correctly).

8.4.4 Aibo ERS-7 model

This model's default object is located in `objects/aibo_ers7.wbt` file. Its VRML structure with various parameter indications and primitive names-to-nodes correspondence can be found in `worlds/aibo_ers7_model.txt`. ERS-7 model schematics are shown in figure 8.8; note that unlike the indicated measurements (provided by Sony), standard Webots units are [m] for position/distance and [rad] for angles.

ERS-7 PIDs primitive IDs are defined in `transfer/openr/include/ers7.h`. Correspondence between common PIDs and their primitive names is listed in table 8.4. Devices specific to the ERS-7 model and their availability are given in table 8.5.

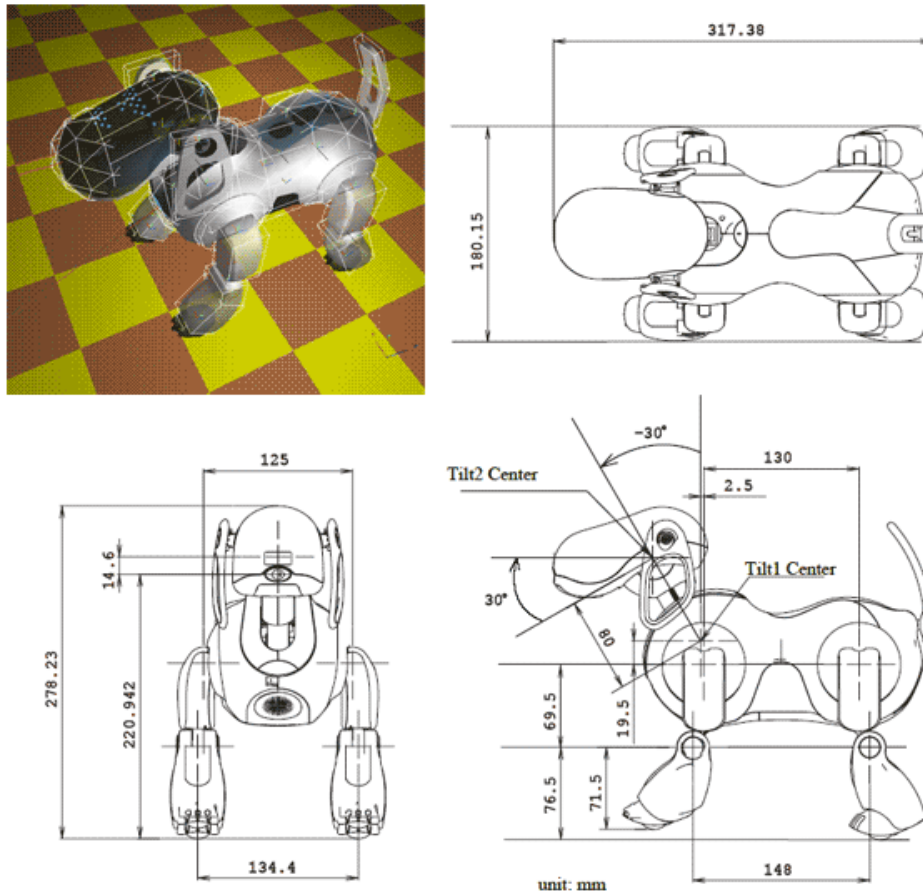


Figure 8.8: ERS-7 model overview (schematics provided by Sony)

Note: For physical details, feedback devices output range, LED positions, face light colors and configurations, etc. please refer to OPEN-R SDK online documentation³, "Model Information for ERS-7".

As you can see, most of ERS-7 devices are implemented in Webots. Let us see which of the ERS-7 *specific* devices are not:

- *Pressure sensors:* like the common pressure sensors, these are useless in simulation and therefore not modelled.

For an explanation of common unsupported devices, please refer to subsection 8.4.2.

Ear plungers

As mentioned previously, plungers are modelled as regular servos whose *min* represents the *off* state, and *max* represents *on*. The ERS-7 has two plungers: the ears. It is of course possible for

³<http://openr.aibo.com>

a controller to command them to any angle between `min` and `max`, however only the extreme positions are meaningful as far as the real robot is concerned.

ERS-7 ears lie "flat" along the sides by default; this is the *off* state. Turning them *on* makes them swing sideways slightly, as if the robot was "lifting" them a bit.

Important: Because of their size, maintaining the ears in the *on* state puts considerable strain on the ear motors; when remotely commanding the real robot, users are therefore cautioned *not* to leave the ears *on* for a long time.

Distance sensors

Aibo ERS-7 model has a distance sensor located in the head of the robot, looking straight ahead, and another in the chest, looking downward at a 60 degree angle.

The head distance sensor is composed of two separate sensors: the *near* and *far* distance sensors, acting (as far as primitive access is concerned) like two separate devices. The *near* sensor has a documented range of 50 cm with a minimal sensing range of 5 cm, while the *far* sensor captures distances between 20 cm and 1.5 meters. The response curve of the model sensors is shown on figure 8.9. Their lookup tables are as follows:

```
"NEAR"
lookupTable [ 0      50  0
              0.05  50  0
              0.275 275 0.1
              0.5   500 0
              ]

"FAR"
lookupTable [ 0      200  0
              0.2    200  0
              0.85   850  0.1
              1.5    1500 0
              ]
```

The chest distance sensor is a regular infrared sensor. Its documented range is 90 cm with a minimal sensing range of 10 cm. The response curve of the chest sensor is shown on figure 8.10. Its lookup table is as follows:

```
lookupTable [ 0      100  0
              0.1    100  0
              0.5    500  0.1
              0.9    900  0
              ]
```

LEDs

None of the ERS-7 model LEDs are actually fully functional: they are only "virtually" supported, i.e., the primitive is accessible like any regular LED device but there will be no *visible* change to the simulated robot (the internal state of the LED is still set appropriately).

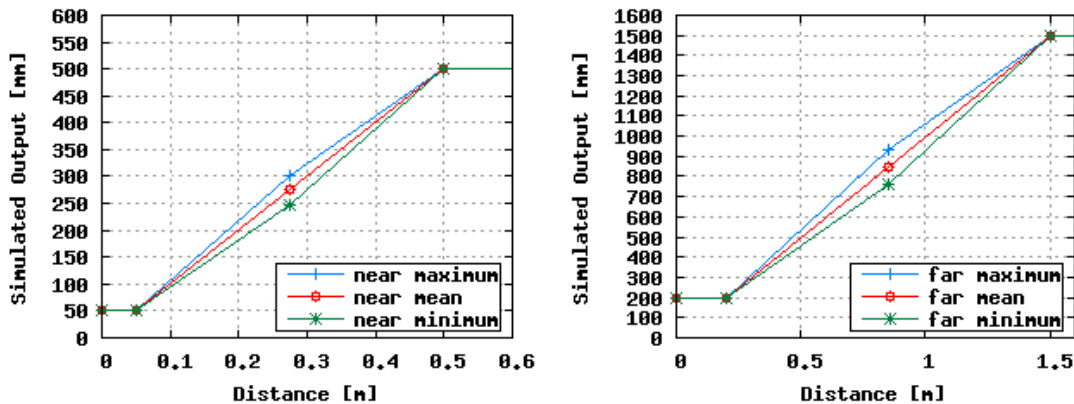


Figure 8.9: Response curves of the ERS-7 *near* (left) and *far* (right) head distance sensors

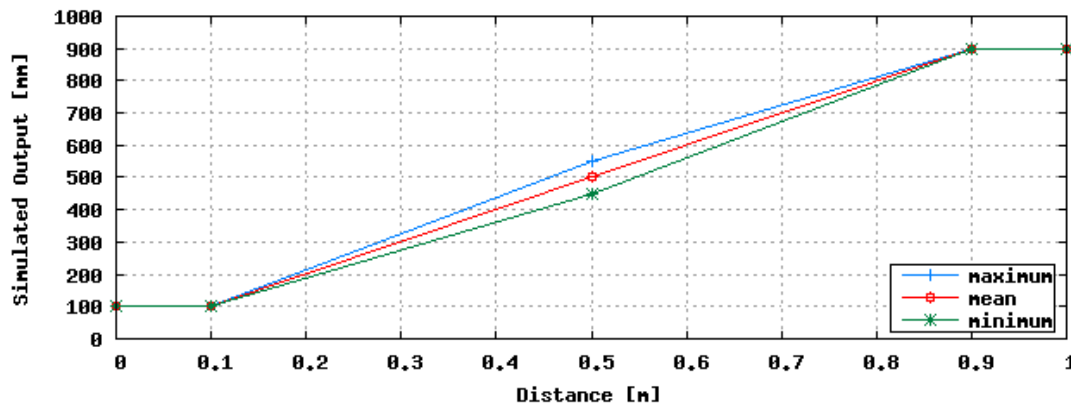


Figure 8.10: Response curve of the ERS-7 chest distance sensor

Face lights have an additional peculiarity: when they are *on*, they can select either of two faces, "A" or "B". All of the face lights select the same face. Face lights form complicated patterns, which may change for any given face light primitive depending on the selected face. Please refer to OPEN-R SDK documentation, "*Model Information for ERS-7*" (section A.2) for details.

Note: Some of the LEDs - face and back lights - also have an *intensity* attribute which has no equivalent in the simulated LED devices; switching them on therefore always corresponds to maximal intensity (minimal intensity being zero, i.e., black).

8.5 Using the Aibo control panel

This section describes the use of the Aibo Control Panel applet integrated into Webots. The Control Panel, shown in figure 8.11, pops up when you double-click on an Aibo robot in a Webots world. It is available for the following models:

- `model="Aibo ERS-210"` (*case-insensitive*): triggers the ERS-210 control panel;
- `model="Aibo ERS-7"` (*case-insensitive*): triggers the ERS-7 control panel.

The model string is obtained from the `model` field of the robot's `CustomRobot` root node.

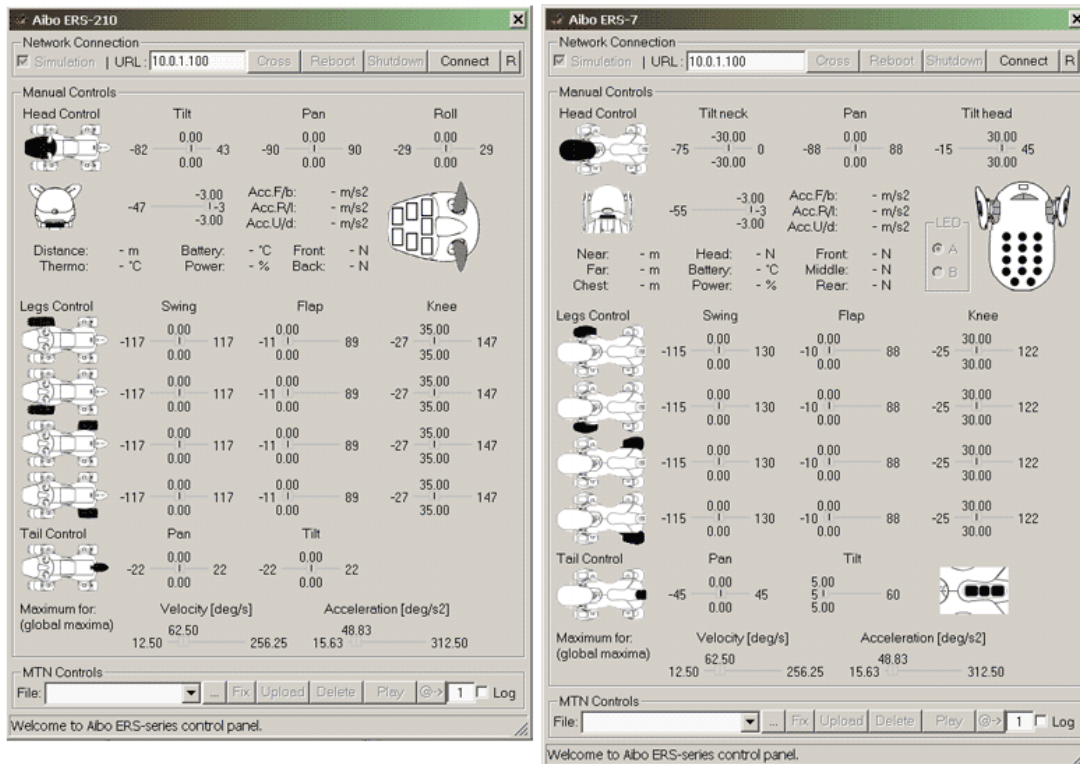


Figure 8.11: Robot windows for the ERS-210 (left) and ERS-7 (right) Aibo models

The title of the Control Panel window may show the name of the robot (if any is defined in the robot's name field), and always shows the exact Aibo model string. The status bar displays useful information such as success or failure of requested operations, error reports, etc.

The Control Panel is divided up into three distinct panes:

- *Remote (network) functions pane*: provides network connection and remote management capabilities for use with a real Aibo robot (running the custom `RCServer` software, see subsection 8.2.4);
- *Manual controls and feedback pane*: provides direct control over Aibo's individual actuators (joints, plungers, LEDs), also serves to display feedback data obtained from simulation or from remote robot, depending on connection state;
- *Motion sequence (MTN) playback pane*: provides management capabilities for the motion sequence (MTN) files for both the simulated and the real robot.

Once the simulation is started or a remote connection established, the various controls can be used to change the state of the simulated or remote robot, e.g., as shown in figure 8.12 for the ERS-210 and in figure 8.13 for the ERS-7 model.

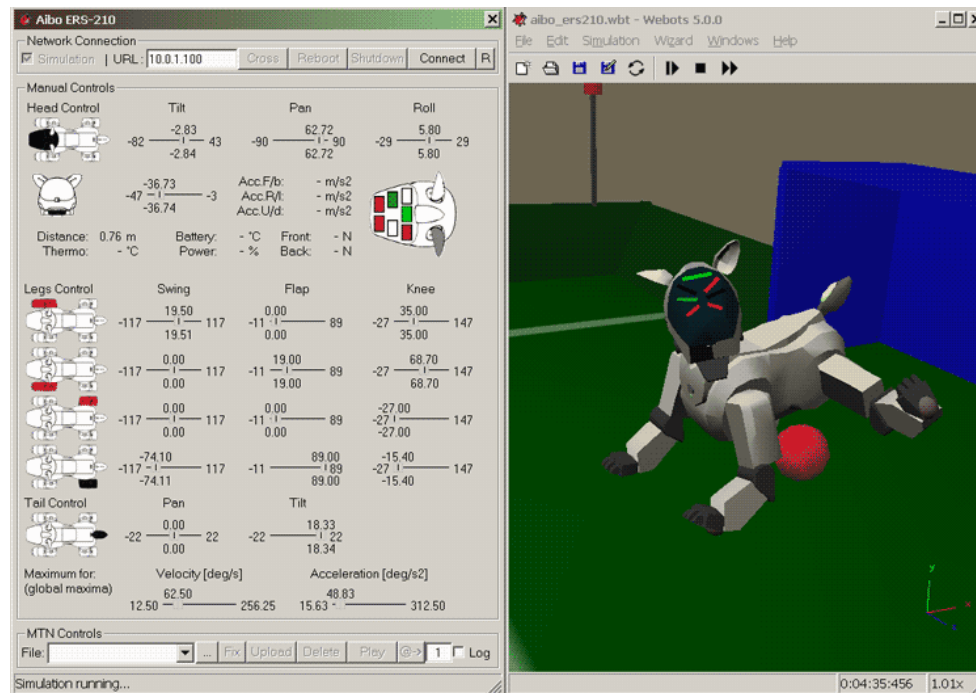


Figure 8.12: Simulated ERS-210 robot after some Control Panel manipulations

Description	ERS-210 PID	Primitive name	Node type
Neck tilt	HEAD_TILT	PRM:/r1/c1-Joint2:j1	servo
Head pan	HEAD_PAN	PRM:/r1/c1/c2-Joint2:j2	servo
Head roll	HEAD_ROLL	PRM:/r1/c1/c2/c3-Joint2:j3	servo
Mouth	HEAD_MOUTH	PRM:/r1/c1/c2/c3/c4-Joint2:j4	servo
Left ear	HEAD_LEFT_EAR	PRM:/r1/c1/c2/c3/e1-Joint3:j5	servo (plunger)
Right ear	HEAD_RIGHT_EAR	PRM:/r1/c1/c2/c3/e2-Joint3:j6	servo (plunger)
Chin sensor	HEAD_CHIN_SWITCH	PRM:/r1/c1/c2/c3/c4/s5-Sensor:s5	-
Color camera	HEAD_CAMERA	...c1/c2/c3/i1-FbkImageSensor:F1	camera
Microphone	HEAD_MIC	PRM:/r1/c1/c2/c3/m1-Mic:M1	-
Speaker	HEAD_SPEAKER	PRM:/r1/c1/c2/c3/s1-Speaker:S1	-
Left front leg, swing	LFLEG_J1	PRM:/r2/c1-Joint2:j1	servo
Left front leg, flap	LFLEG_J2	PRM:/r2/c1/c2-Joint2:j2	servo
Left front leg, knee	LFLEG_J3	PRM:/r2/c1/c2/c3-Joint2:j3	servo
Left front leg, paw	LFLEG_PAW	PRM:/r2/c1/c2/c3/c4-Sensor:s4	touch sensor
Left rear leg, swing	LRLEG_J1	PRM:/r3/c1-Joint2:j1	servo
Left rear leg, flap	LRLEG_J2	PRM:/r3/c1/c2-Joint2:j2	servo
Left rear leg, knee	LRLEG_J3	PRM:/r3/c1/c2/c3-Joint2:j3	servo
Left rear leg, paw	LRLEG_PAW	PRM:/r3/c1/c2/c3/c4-Sensor:s4	touch sensor
Right front leg, swing	RFLEG_J1	PRM:/r4/c1-Joint2:j1	servo
Right front leg, flap	RFLEG_J2	PRM:/r4/c1/c2-Joint2:j2	servo
Right front leg, knee	RFLEG_J3	PRM:/r4/c1/c2/c3-Joint2:j3	servo
Right front leg, paw	RFLEG_PAW	PRM:/r4/c1/c2/c3/c4-Sensor:s4	touch sensor
Right rear leg, swing	RRLEG_J1	PRM:/r5/c1-Joint2:j1	servo
Right rear leg, flap	RRLEG_J2	PRM:/r5/c1/c2-Joint2:j2	servo
Right rear leg, knee	RRLEG_J3	PRM:/r5/c1/c2/c3-Joint2:j3	servo
Right rear leg, paw	RRLEG_PAW	PRM:/r5/c1/c2/c3/c4-Sensor:s4	touch sensor
Tail tilt	TAIL_TILT	PRM:/r6/c2-Joint2:j2	servo
Tail pan	TAIL_PAN	PRM:/r6/c1-Joint2:j1	servo
Acceler. (front-back)	ACCEL_Y	PRM:/a1-Sensor:a1	-
Acceler. (right-left)	ACCEL_X	PRM:/a2-Sensor:a2	-
Acceler. (up-down)	ACCEL_Z	PRM:/a3-Sensor:a3	-

Table 8.2: ERS-210 *common* primitives and their fully-qualified names

Description	ERS-210 PID	Primitive name	Node type
Head pressure (back)	HEAD_SENSOR_BACK	PRM:/r1/c1/c2/c3/f1-Sensor:f1	-
Head pressure (front)	HEAD_SENSOR_FRONT	PRM:/r1/c1/c2/c3/f2-Sensor:f2	-
Distance sensor (PSD)	HEAD_PSD	PRM:/r1/c1/c2/c3/p1-Sensor:p1	distance s.
Eye light (lower left)	HEAD_LED1 (red)	PRM:/r1/c1/c2/c3/l1-LED2:l1	LED
Eye light (middle left)	HEAD_LED2 (green)	PRM:/r1/c1/c2/c3/l2-LED2:l2	LED
Eye light (upper left)	HEAD_LED3 (red)	PRM:/r1/c1/c2/c3/l3-LED2:l3	LED
Eye light (lower right)	HEAD_LED4 (red)	PRM:/r1/c1/c2/c3/l4-LED2:l4	LED
Eye light (middle right)	HEAD_LED5 (green)	PRM:/r1/c1/c2/c3/l5-LED2:l5	LED
Eye light (upper right)	HEAD_LED6 (red)	PRM:/r1/c1/c2/c3/l6-LED2:l6	LED
Mode indicator	HEAD_MODE_INDICATOR	PRM:/r1/c1/c2/c3/l7-LED2:l7	LED
Thermo sensor	TAIL_THERMO	PRM:/r6/t1-Sensor:t1	-
Back pressure sensor	TAIL_SENSOR_BACK	PRM:/r6/s1-Sensor:s1	-
Tail light (blue)	TAIL_LED1 (blue)	PRM:/r6/l1-LED2:l1	LED[*]
Tail light (orange)	TAIL_LED2 (orange)	PRM:/r6/l2-LED2:l2	LED[*]

Table 8.3: ERS-210 *specific* primitives and their fully-qualified names; [*] denotes "virtual" LEDs, i.e., LEDs which lack a proper graphical representation

Description	ERS-7 PID	Primitive name	Node type
Neck tilt	HEAD_TILT1	PRM:/r1/c1-Joint2:11	servo
Head pan	HEAD_PAN	PRM:/r1/c1/c2-Joint2:12	servo
Head tilt	HEAD_TILT2	PRM:/r1/c1/c2/c3-Joint2:13	servo
Mouth	HEAD_MOUTH	PRM:/r1/c1/c2/c3/c4-Joint2:14	servo
Left ear	HEAD_LEFT_EAR	PRM:/r1/c1/c2/c3/e5-Joint3:15	servo (plunger)
Right ear	HEAD_RIGHT_EAR	PRM:/r1/c1/c2/c3/e6-Joint3:16	servo (plunger)
Chin sensor	HEAD_CHIN	PRM:/r1/c1/c2/c3/c4/s5-Sensor:s5	-
Color camera	HEAD_CAMERA	...c1/c2/c3/i1-FbkImageSensor:F1	camera
Stereo microphones	HEAD_MIC	PRM:/r1/c1/c2/c3/m1-Mic:M1	-
Speaker	SPEAKER	PRM:/s1-Speaker:S1	-
Left front leg, swing	LFLEG_J1	PRM:/r2/c1-Joint2:21	servo
Left front leg, flap	LFLEG_J2	PRM:/r2/c1/c2-Joint2:22	servo
Left front leg, knee	LFLEG_J3	PRM:/r2/c1/c2/c3-Joint2:23	servo
Left front leg, paw	LFLEG_PAW	PRM:/r2/c1/c2/c3/c4-Sensor:24	touch sensor
Left rear leg, swing	LRLEG_J1	PRM:/r3/c1-Joint2:31	servo
Left rear leg, flap	LRLEG_J2	PRM:/r3/c1/c2-Joint2:32	servo
Left rear leg, knee	LRLEG_J3	PRM:/r3/c1/c2/c3-Joint2:33	servo
Left rear leg, paw	LRLEG_PAW	PRM:/r3/c1/c2/c3/c4-Sensor:34	touch sensor
Right front leg, swing	RFLEG_J1	PRM:/r4/c1-Joint2:41	servo
Right front leg, flap	RFLEG_J2	PRM:/r4/c1/c2-Joint2:42	servo
Right front leg, knee	RFLEG_J3	PRM:/r4/c1/c2/c3-Joint2:43	servo
Right front leg, paw	RFLEG_PAW	PRM:/r4/c1/c2/c3/c4-Sensor:44	touch sensor
Right rear leg, swing	RRLEG_J1	PRM:/r5/c1-Joint2:51	servo
Right rear leg, flap	RRLEG_J2	PRM:/r5/c1/c2-Joint2:52	servo
Right rear leg, knee	RRLEG_J3	PRM:/r5/c1/c2/c3-Joint2:53	servo
Right rear leg, paw	RRLEG_PAW	PRM:/r5/c1/c2/c3/c4-Sensor:54	touch sensor
Tail tilt	TAIL_TILT	PRM:/r6/c1-Joint2:61	servo
Tail pan	TAIL_PAN	PRM:/r6/c2-Joint2:62	servo
Acceler. (front-back)	ACCEL_Y	PRM:/a1-Sensor:a1	-
Acceler. (right-left)	ACCEL_X	PRM:/a2-Sensor:a2	-
Acceler. (up-down)	ACCEL_Z	PRM:/a3-Sensor:a3	-

Table 8.4: ERS-7 common primitives and their fully-qualified names

Description	ERS-7 PID	Primitive name	Node type
Head pressure sensor	HEAD_SENSOR	PRM:/r1/c1/c2/c3/t1-Sensor:t1	-
Head distance (near)	...CE_SENSOR_NEAR [1]	PRM:/r1/c1/c2/c3/p1-Sensor:p1	distance s.
Head distance (far)	...CE_SENSOR_FAR [1]	PRM:/r1/c1/c2/c3/p2-Sensor:p2	distance s.
Chest distance sensor	CHEST_DISTANCE	PRM:/p1-Sensor:p1	distance s.
Head light (color)	HEAD_LIGHT_COLOR	PRM:/r1/c1/c2/c3/l1-LED2:l1	LED[*]
Head light (white)	HEAD_LIGHT_WHITE	PRM:/r1/c1/c2/c3/l2-LED2:l2	LED[*]
Mode indicator (red)	HEAD_MODE_RED	PRM:/r1/c1/c2/c3/l3-LED2:l3	LED[*]
Mode indicator (green)	HEAD_MODE_GREEN	PRM:/r1/c1/c2/c3/l4-LED2:l4	LED[*]
Mode indicator (blue)	HEAD_MODE_BLUE	PRM:/r1/c1/c2/c3/l5-LED2:l5	LED[*]
Wireless light	HEAD_WIRELESS	PRM:/r1/c1/c2/c3/l6-LED2:l6	LED[*]
Face light 1	HEAD_FACE1	PRM:/r1/c1/c2/c3/l1a-LED3:l1a	LED[*]
Face light 2	HEAD_FACE2	PRM:/r1/c1/c2/c3/l1b-LED3:l1b	LED[*]
Face light 3	HEAD_FACE3	PRM:/r1/c1/c2/c3/l1c-LED3:l1c	LED[*]
Face light 4	HEAD_FACE4	PRM:/r1/c1/c2/c3/l1d-LED3:l1d	LED[*]
Face light 5	HEAD_FACE5	PRM:/r1/c1/c2/c3/l1e-LED3:l1e	LED[*]
Face light 6	HEAD_FACE6	PRM:/r1/c1/c2/c3/l1f-LED3:l1f	LED[*]
Face light 7	HEAD_FACE7	PRM:/r1/c1/c2/c3/l1g-LED3:l1g	LED[*]
Face light 8	HEAD_FACE8	PRM:/r1/c1/c2/c3/l1h-LED3:l1h	LED[*]
Face light 9	HEAD_FACE9	PRM:/r1/c1/c2/c3/l1i-LED3:l1i	LED[*]
Face light 10	HEAD_FACE10	PRM:/r1/c1/c2/c3/l1j-LED3:l1j	LED[*]
Face light 11	HEAD_FACE11	PRM:/r1/c1/c2/c3/l1k-LED3:l1k	LED[*]
Face light 12	HEAD_FACE12	PRM:/r1/c1/c2/c3/l1l-LED3:l1l	LED[*]
Face light 13	HEAD_FACE13	PRM:/r1/c1/c2/c3/l1m-LED3:l1m	LED[*]
Face light 14	HEAD_FACE14	PRM:/r1/c1/c2/c3/l1n-LED3:l1n	LED[*]
Back pressure (rear)	BACK_SENSOR_REAR	PRM:/t2-Sensor:t2	-
Back pressure (middle)	BACK_SENSOR_MIDDLE	PRM:/t3-Sensor:t3	-
Back pressure (front)	BACK_SENSOR_FRONT	PRM:/t4-Sensor:t4	-
Back light (front/color)	BACK_LIGHT_FC	PRM:/l1u-LED3:l1u	LED[*]
Back light (front/white)	BACK_LIGHT_FW	PRM:/l1v-LED3:l1v	LED[*]
Back light (middle/color)	BACK_LIGHT_MC	PRM:/l1w-LED3:l1w	LED[*]
Back light (middle/white)	BACK_LIGHT_MW	PRM:/l1x-LED3:l1x	LED[*]
Back light (rear/color)	BACK_LIGHT_RC	PRM:/l1y-LED3:l1y	LED[*]
Back light (rear/white)	BACK_LIGHT_RW	PRM:/l1z-LED3:l1z	LED[*]

Table 8.5: ERS-7 *specific* primitives and their fully-qualified names; [1] "...CE" expands to "HEAD_DISTANCE"; [*] denotes "virtual" LEDs, i.e., LEDs which lack a proper graphical representation

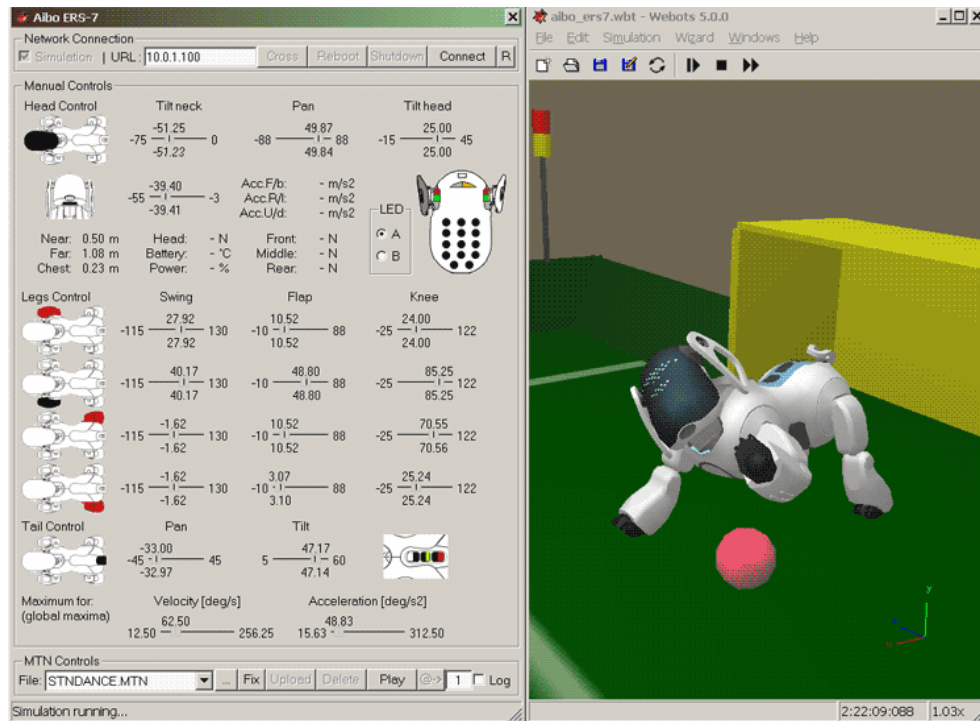


Figure 8.13: Simulated ERS-7 robot after some Control Panel manipulations

One important element to understand is the **Simulation** checkbox. It controls a notion known as *simulation hook*, which is the simulation equivalent to the remote *connected* state:

- **Simulation** checked: simulation is *hooked*, in this mode the Control Panel has control over the simulation; this means, inter alia, that changes to Control Panel active elements (joint sliders, plungers, LEDs) are reflected on the simulation;
- **Simulation** unchecked: simulation is *unhooked*, in this mode there is no link whatsoever between the simulation and the Control Panel; pure remote control is thus possible.

Important: Unless a remote robot is connected to the Control Panel, simulation is forced to hooked state. Once a remote connection is established, it is possible to uncheck the **Simulation** checkbox and use the Control Panel for pure remote control.

The following subsections provide a detailed overview of each pane and the functionalities they offer.

8.5.1 Remote (network) functions

The **Network Connection** pane provides connectivity functions. It is identical for both supported Aibo models. We have already discussed the use of the very first element, the **Simulation** checkbox. Let's see what the other controls do:

- **URL** field: enter the address of your Aibo here; it can be either its IP address (default 10.0.1.100 is entered for you) or its network name (e.g., if you add an alias for Aibo's IP address to your `hosts` system file); click on **Connect** or press `Enter` to initiate the connection;
- **Connect**: initiates the connection with the remote robot; during the connection process, the status bar will report connection progress; once the connection is established, this button changes to **Disconnect**, which serves to close the connection.

The following controls are only available when connected:

- **Cross**: uploads the cross-compiled controller binary to the remote robot (cross-compilation is explained in subsection 8.7.1); the controller binary is expected to be locally found as `OPEN-R/MW/OBJS/CONTROLL.BIN` in the robot's assigned controller directory, and is uploaded through wireless TCP/IP to the `/MS/OPEN-R/MW/OBJS/` location on the Memory stick on the robot;
- **Reboot**: reboots the remote robot; connection is dropped as a result, you must reconnect manually once the robot is back up;
- **Shutdown**: shuts down the remote robot; connection is dropped as a result.

Note: Connection may be automatically dropped if too many transmission errors occur; in that case, the status bar will inform of the disconnection and possibly provide an error report (under Windows, it will also state the last system error message).

Important: If Aibo is physically switched off before disconnecting, the Control Panel may freeze for a while, causing Webots to become unavailable or not responding. It is therefore always better to disconnect the robot from the Control Panel before switching it off.

8.5.2 Manual controls and feedback

The **Manual Controls** pane provides manual control functionality over the simulated or remote robot, depending on the connection state, and also serves to display sensor feedback data. It is essentially identical for both supported Aibo models.

Unless otherwise indicated, each element of this section corresponds to a robot primitive. Some primitives, e.g., pressure sensors, may have several elements associated to them. In this section, we refer to those primitives by their descriptive names. Please check section 8.4 for a list of primitive names and their corresponding PIDs.

Note: Manual controls are disabled when manual control is not possible, i.e., either during MTN playback or when the simulation is stopped and no remote connection has been established. Disabled state is recognizable from the sliders' grey appearance.

Head controls

The **Head Control** section contains the following elements (“*” denotes *active* controls):

- *head top view*: body picture with selected head; serves also to display certain touch sensors state (see below);
- *head joint sliders* *: command the 3 head joint primitives (available for both models; some may have different semantics depending on the model, please refer to slider labels);
- *head front view*: head picture with selected mouth; serves also to display certain touch sensors state (see below);
- *mouth joint slider* *: commands the mouth joint primitive (available for both models);
- *numeric data fields*: display numeric feedback data obtained from various sensors (see below);
- *face view* *: face picture with various active elements: ears, face lights, LEDs (discussed separately for each supported model).

Available *active* controls, marked with “*”, are used to command various parts of the Aibo robot. If a remote connection has been established, these commands are sent to the remote robot. If the simulation is hooked, they also get processed by the simulation engine. The remaining passive elements serve to display robot status. The following graphical displays are common to both Aibo models:

- *Head touch sensor(s)*: displayed on the head top view; when pressure is detected on either head sensor, part of the selected head turns red;
- *Chin touch sensor*: displayed on the head front view; when pressure is detected, the selected chin turns red;
- *Back touch sensor(s)*: displayed on the head top view; when pressure is detected, the corresponding part on the back of the robot turns red.

Some numeric data displays are also common to both models:

- **Acc.F/b** [m/s²]: front-back acceleration (front positive);
- **Acc.R/l** [m/s²]: right-left acceleration (right positive);
- **Acc.U/d** [m/s²]: up-down acceleration (up positive);
- **Battery** [C]: battery temperature, measured in degrees Celsius;
- **Power** [%]: remaining battery power, measured in percent of total battery capacity.

Additional model-specific head controls are discussed in the following subsections.

Head controls: ERS-210 specifics

The ERS-210 face view (see figure 8.14) contains the following active controls:

- *6 face lights*: these are represented by the 6 rectangles located along the edges of the face; they are white when lights are off, and turn to appropriate color when they are on;
- *mode indicator*: represented by the one remaining rectangle located in the middle of the face; it is also white when the mode indicator is off, and turns green when it is on;
- *ears* : ear plungers are controlled by two active ear-shaped elements; ears are pricked when the ear elements are active (grey), and flat when they are inactive (white).

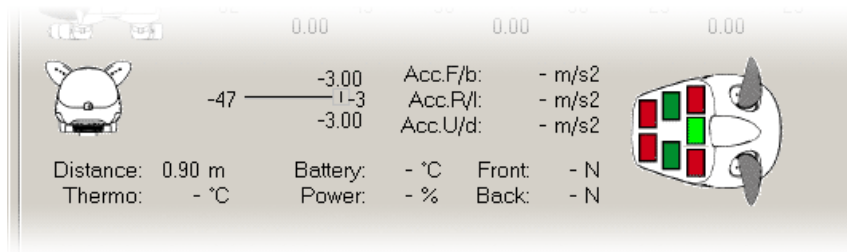


Figure 8.14: ERS-210 face and data view (lights and ears switched on)

Additionally, the ERS-210 model has the following proper numeric data displays:

- **Distance** [m]: head distance sensor (PSD);
- **Thermo** [C]: temperature sensor;
- **Front** [N]: head pressure sensor (front);
- **Back** [N]: head pressure sensor (back);

Head controls: ERS-7 specifics

The ERS-7 face view (see figure 8.15) contains the following active controls:

- *14 face lights*: these are represented by the 14 circles filling up the face region, which are an abstraction of the 14 primitives making up the face lights, numbered 1 to 14 from left to right and from top to bottom; they are black when off, and turn white when switched on, regardless of LED specs;
- *head light* : represented by the 2 shapes forming a semi-circle in the top-center of the face, which are an abstraction for the white (leftmost) and color (rightmost) LEDs making up the head light; they both appear grey when off, and turn white and orange respectively when switched on;

- *mode indicator* : represented by the 3 pairs of mirrored shapes forming up two semi-circles located by each ear, which are an abstraction for the green (topmost), red (middle) and blue (downmost) LEDs making up the mode indicator; they appear grey when off, and turn appropriate color when switched on;
- *wireless light* : represented by the small rectangle located at the very top of the face; it is white when off, and turns green when on;
- *ears* : ear plungers are controlled by two active ear-shaped elements; ears are raised when the ear elements are active (grey), and flat when they are inactive (white).

Note: Head and face lights and LEDs have no hooks on the simulated robot model. What appears to be face lights or other LEDs on the model are simple static graphics. This may be corrected when the ERS-7 model is updated in a future release.

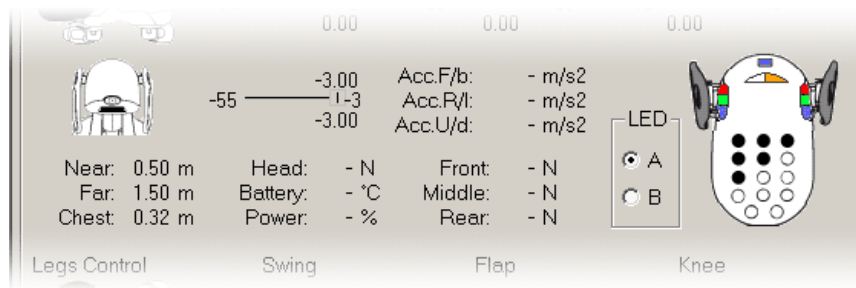


Figure 8.15: ERS-7 face and data view (some lights and ears switched on)

Additionally, the ERS-7 model has the following proper numeric data displays:

- **Near** [m]: head distance sensor (near);
- **Far** [m]: head distance sensor (far);
- **Chest** [m]: chest distance sensor;
- **Head** [N]: head pressure sensor;
- **Front** [N]: back pressure sensor (front);
- **Middle** [N]: back pressure sensor (middle);
- **Rear** [N]: back pressure sensor (rear);

Legs controls

The **Legs Control** section is identical for both supported models. It contains 4 rows of joint controls, one row for each leg (front legs on top, hind legs on bottom). Each row is made up of the following elements:

- *a robot view* : top robot view where selected paw indicates leg position; that paw also turns red when the corresponding paw touch sensor is triggered;
- *3 joint sliders* : these correspond to the 3 controllable leg joints for selected leg, i.e., swing (J1), flap (J2) and knee (J3).

Tail controls

The **Tail Control** section is identical for both supported models. It contains 2 joint sliders controlling the pan and tilt of the robot's tail.

The ERS-7 model also has back lights control, described below.

Back lights control (ERS-7 only)

The ERS-7 model has an additional control element, shown in figure 8.16: a view of the back of the robot, located to the right of the **Tail Control**, containing 6 active elements placed in a cluster in the center. These active elements, arranged in a row from left to right, represent 6 LEDs which are an abstraction for the 3 ERS-7 back lights:

- *front light* : the first pair of elements represent the 2 LEDs making up the front back light; they appear black when turned off, and change to white (leftmost) or blue (rightmost) when switched on;
- *middle light* : the next pair of elements represent the 2 LEDs making up the middle back light; they appear black when turned off, and change to white (leftmost) or yellow (rightmost) when switched on;
- *rear light* : the last pair of elements represent the 2 LEDs making up the rear back light; they appear black when turned off, and change to white (leftmost) or red (rightmost) when switched on.

Joint parameters controls

This section, labelled **Maximum for...**, is identical for both supported models. It contains a set of 2 sliders:

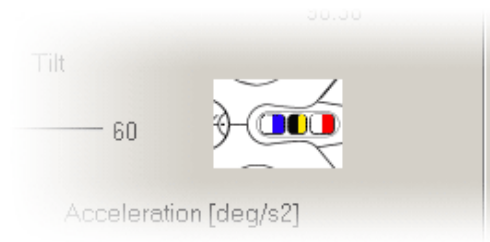


Figure 8.16: ERS-7 back lights control ('white' middle LED is off)

- **Velocity** [deg/s] : controls the global maximum setting for joint velocity;
- **Acceleration** [deg/s²] : controls the global maximum setting for joint acceleration.

Note: Changing the value of either slider affects all joints. Individual joint limits are however still enforced; each joint's commanded parameter (velocity or acceleration) will be set as close to the requested value as individual limits permit.

8.5.3 Motion sequence (MTN) playback

The **MTN Controls** pane provides motion sequence (*MTN*) files management and playback functions. It is identical for both supported Aibo models. MTN is Sony's format for frame-by-frame motion sequence playback. MTN files are binary files expected to have `.mtn` extension and be stored in 8.3 filename format. For further information regarding the MTN format, please refer to the official Aibo SDE website⁴.

Let's see what the available MTN controls do:

- **File** field : displays the selected MTN file name (the combo box contains a list of previously selected file names); to select a new file, click on the [...] button located next to the combobox;
- **Fix** : attempts to *fix* the selected MTN file (see notes below); the resulting file replaces the original, which is first renamed for backup;
- **Play** : starts the selected MTN file playback; if a remote robot is connected, then a command is sent to start remote MTN playback (the selected MTN file must be on the robot's Memory stick), otherwise MTN is played in simulation; if playback is successfully started, the button changes to **Stop**, which aborts the playback;
- **@->** button : updates the number of times to repeat current MTN motion sequence playback to the desired value (only available during playback);

⁴<http://openr.aibo.com>

- **@->** field : enter the number of times to repeat current MTN motion sequence here (any number between 0 and 65535 can be used); pressing **Enter** has the effect of the **@->** button during playback and that of the **Play** button otherwise (in that case, minimum value is 1);
- **Log** checkbox : enables or disables the logging of MTN playback; log file is written to Webots **User directory**, in append mode; it is a binary file which starts with the "AIBO_2.0" string, followed by logged MTN data for each keyframe, as follows:
 - (a) *simulation feedback data* : number n of following values (1 byte) + mtm joint values (m shorts) + paw touch sensor values (4 shorts);
 - (b) *keyframe data* : keyframe index (1 short) + keyframe mtm joint values (m shorts) from the MTN file being played;
 - (c) *remote feedback data* : number n of following values (1 byte) + mtm joint values (m shorts) + paw touch sensor values (4 shorts)

(short integers are in little-endian encoding; data length: $m=15$ for ERS-210 or $m=20$ for ERS-7; for feedback data, number of values n may be 0 if no data is available, otherwise $n=m+4$).

The remaining MTN file management controls are only available when connected (and not during MTN playback):

- **Upload** : uploads the selected MTN file to the remote robot; the file is uploaded to the /MS/OPEN-R/MW/DATA/P/ directory on the robot's Memory stick; if a file with that name already exists, it is overwritten without prompting;
- **Delete** : deletes the remote file with selected filename from the /MS/OPEN-R/MW/DATA/P/ directory on robot's Memory stick; local selected file is not affected.

During MTN playback, **Manual Controls** are disabled, and the status bar indicates the motion sequence progress, i.e., the current keyframe index followed by the total number of frames, and optionally the number of repeat loops yet to play. Example (also shown in figure 8.17):

```
Simulated MTN running frame 57/72 (2x more)...
```

In the above example, the status bar tells us that the MTN playback is being simulated, that we have reached keyframe 57 of a total of 72, and that the whole sequence will be replayed 2 more times.

Here's another example:

```
Remote MTN running frame 1433/1872...
```

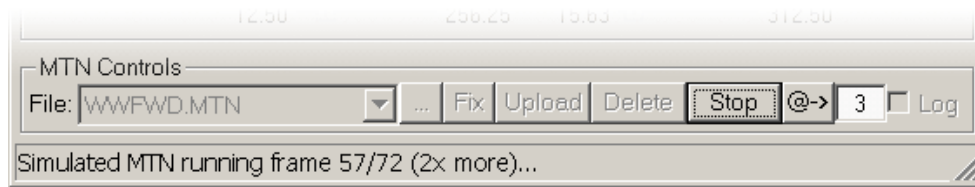


Figure 8.17: Status bar shows progress info during MTN playback

In that example, the MTN motion sequence is being played on the remote robot, who has reached keyframe 1433 of a total of 1872, and there are no scheduled replays.

Note: You can modify the number of scheduled replays at any time during playback by entering the desired number in the @-> field and clicking on the @-> or simply pressing **Enter**. Enter 0 to clear scheduled replays. (This will *not* stop the playback. Use the **Stop** button to that effect.)

MTN support limitations and the *fix* feature

There are some limitations to the MTN motion sequences supported by our software, which are mostly due to the remote `RCServer` software MTN support limitations. The MTN *fix* feature can be used to adapt foreign MTN motion sequences to our software requirements. It will, *inter alia*:

- amend the design label to the correct one for current model;
- re-order primitive data in keyframes to match hard-coded MTN primitives list;
- fill data for missing MTN primitives with default values and discard data for primitives which do not appear in the hard-coded MTN primitives list;
- expand the interpolation between keyframes using linear interpolation algorithm so that there are no keyframes with a positive interpolation factor.

The MTN *fix* feature may also be used to help convert MTN files written for one model to another model. In order to do that, follow these few simple steps:

1. Edit a *copy* of the original MTN file.
2. Change old model primitive names, which appear in plain text in the binary MTN file, to their new model counterparts. You needn't worry about primitives which have no correspondance in the new model: data for unknown primitives will simply be discarded.
3. Finally, open the resulting MTN file in the *new model's* Control Panel and **Fix** it.

(You may refer to `worlds/aibo_ers7ers210_prm_map.txt` for reference to primitive names correspondance between ERS-7 and ERS-210 models.)

Important: If you need to *suppress* characters from a primitive name, you need only set the byte following the shortened primitive name to 00. If you need to *add* characters, however, you must also amend the name's length (stored on the preceding byte) as well as the total size of the primitive names data section (stored in little-endian encoding on 4 bytes at the beginning of the section; there are 2 additional bytes between the section length and the first primitive name length, which you must not touch).

8.6 Programming your Aibo

8.6.1 Getting started

We assume here that you are already familiar with basic Webots controller programming techniques. If you are not, you are encouraged to go through the tutorial in chapter 3. For documentation pertaining to the controller API function calls, please refer to the *Reference Manual*.

To write a simple Aibo program, put it in a subdirectory named after your controller in your `controllers/` user directory, then use the following `Makefile` to compile it:

```
# for Windows, uncomment following line if you need the
# DOS console to appear, e.g., for console output
#DOS_CONSOLE=1
include ../Makefile.include
```

This works if your controller source has the same name as the subdirectory it is in. (The language is selected automatically from the source extension: `.c` for C, `.cpp/.cc` for C++.) If your controller uses several source files, you must change above `Makefile` accordingly (see `Makefile.include` for available options).

Aibo programming basics

Aibo's devices are identified by the corresponding primitive names. As an example, we provide a listing of a simple controller, `ersX_camera`, which enables the camera and refreshes its display every 64 milliseconds. This controller, actually valid for both supported models, is contained in `controllers/ersX_camera/ersX_camera.c`:

```
#include <device/robot.h>
#include <device/camera.h>
```

```

/* this very simple controller enables the camera and refreshes
 * its display every 64 milliseconds; it is equally useable for
 * Aibo ERS-210 and ERS-7 robots, since the camera primitive name
 * is identical for both models */

#define SIMULATION_STEP 64
static DeviceTag camera;

static void init(void) {
    camera = robot_get_device("PRM:/r1/c1/c2/c3/i1-FbkImageSensor:F1");
    camera_enable(camera, SIMULATION_STEP);
    /* note: considerably slows down simulation */
}

static int run(int ms) {
    (void)camera_get_image(camera); /* refresh camera image */
    return SIMULATION_STEP;
}

int main(void) {
    robot_live(init);
    robot_run(run); /* note: never returns */
    return 0;
}

```

Using PID constants to access devices

To avoid retyping primitive names, we provide two include files which define PID constants and primitive name tables for each model (see section 8.4). These files are located in `Webots transfer/openr/aibo/` directory. Copy the `ers210.h` or `ers7.h` file from that directory to your `controllers/` user subdirectory and use the following include includes in your controller's source:

```
#include "../ers210.h"
```

for the ERS-210, or

```
#include "../ers7.h"
```

for the ERS-7 model. You will thus be able to use PID constants to address the desired primitives, whose names are stored for both models in the following array:

```

static const char* const PRIMITIVE_LOCATOR[]; // prm names, indexed by PID
static const int NUM_PRIMITIVES;           // size of above array

```

Aibo's devices can now be easily referenced using their PID constants. The simple controller whose source is given in previous section now becomes:

```
#include <device/robot.h>
#include <device/camera.h>
#include "../ers7.h"
/* model-specific additional include: this controller is now specific
 * to the ERS-7 model (include "../ers210.h" for ERS-210 model) */

#define SIMULATION_STEP 64
static DeviceTag camera;

#define PRM(pid) PRIMITIVE_LOCATOR[pid]
/* macro for ease of referencing device names */

static void reset(void) {
    camera = robot_get_device(PRM(CAMERA));
    camera_enable(camera, SIMULATION_STEP);
}

/* the rest of the code does not change */

static int run(int ms) {
    (void)camera_get_image(camera); /* refresh camera image */
    return SIMULATION_STEP;
}

int main() {
    robot_live(init);
    robot_run(run); /* note: never returns */
    return 0;
}
```

Note: The above code is now specific to the ERS-7 model, because of the model-specific include, which may seem a drawback. For more complicated controllers, however, primitive names for required devices may not be the same for the two models anyway (such as leg joints, for instance), but because PID constant naming is quite consistent, it is much easier, using this system, to adapt the controller from one model to another - just use appropriate model-specific include - whereas otherwise you would need to check and adapt every primitive name used in your source code.

In the following samples, we will use explicit primitive naming for ease of reading. However, they can easily be adapted to the use of PID constants as described above.

8.6.2 Default controllers

The default controllers are known as `ers210` for ERS-210 model and `ers7` for ERS-7 model. They are essentially identical, except of course for the primitive names and number of distance sensors which were adapted for each model as appropriate. Here is the listing for `ers210` controller source, contained in `controllers/ers210/ers210.c`:

```
#include <stdio.h>
#include <string.h>
#include <device/robot.h>
#include <device/servo.h>
#include <device/camera.h>
#include <device/distance_sensor.h>
#include <device/touch_sensor.h>
#include <device/mtn.h>

#define SIMULATION_STEP 64
#define MTN_PATH "../data/mtn/ers210/"
#define MTN_FILE "WWFWD.MTN"
#define MTN_REPLAY 3

static int demo;
/* true => do nothing, false => play mtn; demo mode is
 * triggered by robot's name: "demo" */

static MTN *mtn;
static DeviceTag camera,
    distance_sensor,
    touch_sensor_fore_l,
    touch_sensor_fore_r,
    touch_sensor_hind_l,
    touch_sensor_hind_r;

static void init(void) {
    camera=robot_get_device("PRM:/r1/c1/c2/c3/i1-FbkImageSensor:F1");
    distance_sensor=robot_get_device("PRM:/r1/c1/c2/c3/p1-Sensor:p1");
    touch_sensor_fore_l=robot_get_device("PRM:/r2/c1/c2/c3/c4-Sensor:s4");
    touch_sensor_hind_l=robot_get_device("PRM:/r3/c1/c2/c3/c4-Sensor:s4");
    touch_sensor_fore_r=robot_get_device("PRM:/r4/c1/c2/c3/c4-Sensor:s4");
    touch_sensor_hind_r=robot_get_device("PRM:/r5/c1/c2/c3/c4-Sensor:s4");
    /* demo mode? */
    demo = (strcmp(robot_get_name(),"demo")==0) ? 1 : 0;
    if( !demo ) {
        /* enable camera and sensors */
        camera_enable(camera,SIMULATION_STEP);
        distance_sensor_enable(distance_sensor,SIMULATION_STEP);
    }
}
```

```

touch_sensor_enable(touch_sensor_fore_l, SIMULATION_STEP);
touch_sensor_enable(touch_sensor_hind_l, SIMULATION_STEP);
touch_sensor_enable(touch_sensor_fore_r, SIMULATION_STEP);
touch_sensor_enable(touch_sensor_hind_r, SIMULATION_STEP);
/* read MTN motion sequence */
if( !(mtn = mtn_new(MTN_PATH MTN_FILE)) )
    printf("MTN Error: %s\n", mtn_get_error());
}
}

static void die(void) {
    if( mtn ) mtn_delete(mtn);
}

static int run(int ms) {
    static int loop=-1;
    if( !demo ) {
        (void)camera_get_image(camera); /* refresh camera */
        if( mtn_is_over(mtn) && (++loop<MTN_REPLAY) )
            mtn_play(mtn); /* play mtn until enough loops */
    }
    return SIMULATION_STEP;
}

int main() {
    robot_live(init);
    robot_die(die);
    robot_run(run);
    return 0;
}

```

In the above example, inspired by source code written by Lukas Hohl, the `init()` function first retrieves camera and sensors device tags. We then select *demo* mode if the robot's name="demo" (in this mode, the controller actually does nothing, so as not to interfere with the Control Panel). Finally, unless demo mode is selected, we enable camera and sensors and read the appropriate MTN file from disk.

In the `run()` function, which gets called regularly by the simulation engine, unless we are running in *demo* mode we refresh the camera display and start, or re-start, MTN playback until we reach the number of loops specified by the `MTN_REPLAY` constant.

Note: Sensor readings are not used in this example; sensor-related code is included solely as an example of manipulations required in order to enable their use.

8.6.3 Sample controller: MTN Playlist

A bit more complicated sample controller, known as `ers7.mtn` for ERS-7 and `ers210.mtn` for ERS-210 model, allows you to play custom MTN playlists without recompiling each time you want the robot to execute a different motion sequence. It will read an MTN playlist definition file (see relevant subsection below) and playback the MTN files from the playlist, one after the other.

Source code

Let's take a look at this controller's source code (same controller is also available for the ERS-210 model), contained in `controllers/ers7_mtn/ers7_mtn.cc`:

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <device/robot.h>
4  #include <device/camera.h>
5  #include <device/touch_sensor.h>
6  #include "mtnplaylist.hh"
7
8  #define SIMULATION_STEP 64
9  #define MTN_PLAYLIST "playlist.txt"
10
11 static MTNPlayback *playlist;
12 static DeviceTag camera;
13 static DeviceTag touch_sensor[4];
14 enum { // touch sensor indices
15     TSFL, // front left
16     TSHL, // hind left
17     TSFR, // front right
18     TSHR, // hind right
19     NUM_TS}; // number of touch sensors
20
21
22 static void init(void) {
23     camera = robot_get_device("PRM:/r1/c1/c2/c3/i1-FbkImageSensor:F1");
24     touch_sensor[TSFL]=robot_get_device("PRM:/r2/c1/c2/c3/c4-Sensor:24");
25     touch_sensor[TSHL]=robot_get_device("PRM:/r3/c1/c2/c3/c4-Sensor:34");
26     touch_sensor[TSFR]=robot_get_device("PRM:/r4/c1/c2/c3/c4-Sensor:44");
27     touch_sensor[TSHR]=robot_get_device("PRM:/r5/c1/c2/c3/c4-Sensor:54");
28     /* enable sensors */
29     camera_enable(camera,SIMULATION_STEP);
30     for(int i=0; i<NUM_TS; i++)
31         touch_sensor_enable(touch_sensor[i],SIMULATION_STEP);

```

```

32  /* read playlist */
33  playlist = mtn_playlist_new(MTN_PLAYLIST,stdout); // log to stdout
34  }
35
36  static void die(void) {
37      mtn_playlist_delete(playlist);
38  }
39
40  static void print_touch_sensors_if_changed() {
41      static unsigned short stored_sensor_value[4];
42      static bool first_time = true;
43      /* read sensors values and detect change */
44      bool changed = false;
45      for(int i=0; i<NUM_TS; i++) {
46          unsigned short v = touch_sensor_get_value(touch_sensor[i]);
47          changed |= (stored_sensor_value[i]!=v);
48          stored_sensor_value[i] = v;
49      }
50      /* print sensor values if change detected */
51      if( changed || first_time ) {
52          printf("Touch: ");
53          printf("fore:[%u-%u] ",sensor_value[TSFL],sensor_value[TSFR]);
54          printf("hind:[%u-%u] ",sensor_value[TSHL],sensor_value[TSHR]);
55          printf("\n");
56      }
57      first_time = false;
58  }
59
60  static int run(int ms) {
61      (void)camera_get_image(camera); // refresh cam image
62      print_touch_sensors_if_changed();
63      if( mtn_playlist ) // playback mtn list
64          mtn_playback(mtn_playlist);
65      return SIMULATION_STEP;
66  }
67
68  int main() {
69      robot_live(init);
70      robot_die(die);
71      robot_run(run);
72      return 0;
73  }

```

There is not much difference between this controller and its simpler sibling described in subsection 8.6.2, so we shall skip the basics. The only thing of notice is the creation of an `MTNPlay`

structure, defined in `mtnplaylist.hh`, declared at line 11 and created at line 33 with a call to `mtn_playlist_new()` function.

To make this controller a bit more interesting, the `print_touch_sensors_if_changed()` function, which gets called at every simulation cycle from the `run()` function at line 62, will produce a printout of paw touch sensors' state, but only if at least one value has changed (otherwise, the console would be flooded with largely repeating data).

Notice also the use of an array in conjunction with an enumerated sequence (lines 13-19) to store paw touch sensor device tags. The array makes it easier to enumerate all sensors when required (e.g., lines 30-31), and the `enum` constants allow to easily identify each individual sensor (e.g., lines 53-54).

Note: Included `mtnplaylist.hh` file is located in the same directory as the controller's source. It contains declarations and function definitions for MTN playlist support. It is model- and robot-independent, and can be re-used as is.

Playlist file format

Here is a sample playlist definition file for the ERS-7 model, stored in `playlist.txt` in the controller's directory:

```
#-----
# MTN Playlist Sample
#-----

@../../data/mtn/ers7/
# '@' lead-in, possibly followed by filename prefix (e.g., root path)

2 # number of entries

# entries one per line, as: [filename] [loop] #rest ignored
WWFWD.MTN 5
#WSSDK.MTN 1 # commented out
WWBWD.MTN 3

# after declared number of entries are read,
# everything else is ignored
```

The format is quite simple: The first uncommented non-empty line must contain the '@' lead-in character, possibly followed by a prefix string (no end-of-line comments allowed here) which will be prepended to each of the filename entries that follow; this prefix can be, e.g., the root path for your MTN files. Then, the number of MTN file entries must be specified. Finally, said number of entries must follow, each entry on a separate line (commented-out lines are ignored),

made up of an MTN file name (to which the above prefix is automatically prepended) and the number of times that file is to be played (end-of-line comments are allowed).

The above sample playlist will use MTN files from default MTN storage directory, and make the robot first walk forward 5 times (`WWFWD.MTN`), then backward 3 times (`WWBWD.MTN`).

Note: A playlist is only loosely model-dependent: assuming that valid MTNs with the same names exist for the other model, the only thing to change in order to use this playlist with the other model's MTN playlist controller is the root path line.

Building the controller

To build this controller, the following Makefile is used:

```
SOURCE_INCLUDES = mtnplaylist.hh
DOS_CONSOLE=1
include ../Makefile.include
```

The `SOURCE_INCLUDES` variable is used to declare source code include dependencies, and `DOS_CONSOLE=1` makes a console appear under Windows to display controller's standard output.

8.6.4 Sample controller: Mimic

This controller, called `ers7_mimic` for the ERS-7 and `ers210_mimic` for the ERS-210 model, was invented by Lukas Hohl. Here is the listing for the ERS-210 version source, contained in `controllers/ers210_mimic/ers210_mimic.c`:

```
#include <device/robot.h>
#include <device/servo.h>

/* this controller is quite entertaining: releasing the servos on one
 * of the robot's legs so that it hangs loose, it makes the remaining
 * legs 'mimic' that one as it gets rotated or bent by hand (idea
 * courtesy of Lukas Hohl) */

#define SIMULATION_STEP 16
#define NUM_JOINTS_PER_LEG 3
enum { /* leg indices enumeration */
    LFLEG, /* left fore leg */
    LHLEG, /* left hind leg */
    RFLEG, /* right fore leg */
    RHLEG, /* right hind leg */
    NUM_LEGS};
```

```

#define MASTER_LEG RFLEG /* master leg */
static DeviceTag leg_servos[NUM_LEGS][NUM_JOINTS_PER_LEG];

static void init(void) {
    int j;
    leg_servos[LFLEG][0]=robot_get_device("PRM:/r2/c1-Joint2:j1");
    leg_servos[LFLEG][1]=robot_get_device("PRM:/r2/c1/c2-Joint2:j2");
    leg_servos[LFLEG][2]=robot_get_device("PRM:/r2/c1/c2/c3-Joint2:j3");
    leg_servos[LHLEG][0]=robot_get_device("PRM:/r3/c1-Joint2:j1");
    leg_servos[LHLEG][1]=robot_get_device("PRM:/r3/c1/c2-Joint2:j2");
    leg_servos[LHLEG][2]=robot_get_device("PRM:/r3/c1/c2/c3-Joint2:j3");
    leg_servos[RFLEG][0]=robot_get_device("PRM:/r4/c1-Joint2:j1");
    leg_servos[RFLEG][1]=robot_get_device("PRM:/r4/c1/c2-Joint2:j2");
    leg_servos[RFLEG][2]=robot_get_device("PRM:/r4/c1/c2/c3-Joint2:j3");
    leg_servos[RHLEG][0]=robot_get_device("PRM:/r5/c1-Joint2:j1");
    leg_servos[RHLEG][1]=robot_get_device("PRM:/r5/c1/c2-Joint2:j2");
    leg_servos[RHLEG][2]=robot_get_device("PRM:/r5/c1/c2/c3-Joint2:j3");
    /* release master leg servos, enable position reading */
    for(j=0; j<NUM_JOINTS_PER_LEG; j++) {
        servo_motor_off(leg_servos[MASTER_LEG][j]);
        servo_enable_position(leg_servos[MASTER_LEG][j],SIMULATION_STEP);
    }
}

static int run(int ms) {
    int i,j; float master;
    for(j=0; j<NUM_JOINTS_PER_LEG; j++) {
        /* master position */
        master = servo_get_position(leg_servos[MASTER_LEG][j]);
        for(i=0; i<NUM_LEGS; i++) if (i!=MASTER_LEG)
            servo_set_position(leg_servos[i][j],master);
        /* set remaining legs to master */
    }
    return SIMULATION_STEP;
}

int main(void) {
    robot_live(init);
    robot_run(run);
    return 0;
}

```

In the `init()` function, we begin by obtaining relevant device tags, then release all joint servos on the master leg, defined by the `MASTER_LEG` constant (in this case, it is the robot's *right fore*

leg), and enable their position reading. The `run()` function then simply reads the master position from the master leg servos for each joint and commands the other legs to reach that position. (Notice how we loop on all *joints* first, so as to minimize master servo position readings.)

Note: As you can see, this code is extremely simple yet the effects are quite impressive! You can actually use it in simulation as well as on the real robot: open the Control Panel and try playing with the right fore leg sliders (first row in the **Legs Control** section)... Have fun!

8.7 Transfer to real robot

In this section, we address the question of how to use your controller written using the Webots controller API with your real, live Aibo robot. There are typically two ways to do that:

- *cross-compilation* mode, wherein the controller code is cross-compiled to produce a binary executable which then runs on the live robot directly;
- *remote-control* mode, wherein the controller runs as part of Webots simulation engine but commands are forwarded to and feedback obtained from the live robot by means of a remote (wireless) connection, effectively emulating direct remote execution.

For now, only the cross-compilation method is supported. Future developments may address this issue and extend Aibo support to enable the use of other transfer modes.

8.7.1 Cross-compilation

Software for the Aibo robot is written using Sony's proprietary object-oriented API called OPEN-R. What we have is some source code for a controller written in C or C++ using Webots controller API. The basic idea of cross-compilation is this: an OPEN-R wrapper object running on Aibo basically translates all Webots robot controller API calls into OPEN-R meaningful instructions for the live robot. Because OPEN-R programs are written in C++, it is actually rather straightforward to combine the Webots controller program with the OPEN-R wrapper object to obtain a binary code executable on the live robot. Schematics of this procedure are shown in figure 8.18.

Important: The actual cross-compiler is Sony's OPEN-R cross-compiler included in the OPEN-R SDK available for download from the official OPEN-R website⁵. In order to take advantage of controller cross-compilation in Webots, the OPEN-R SDK must therefore be installed first.

⁵<http://openr.aibo.com>

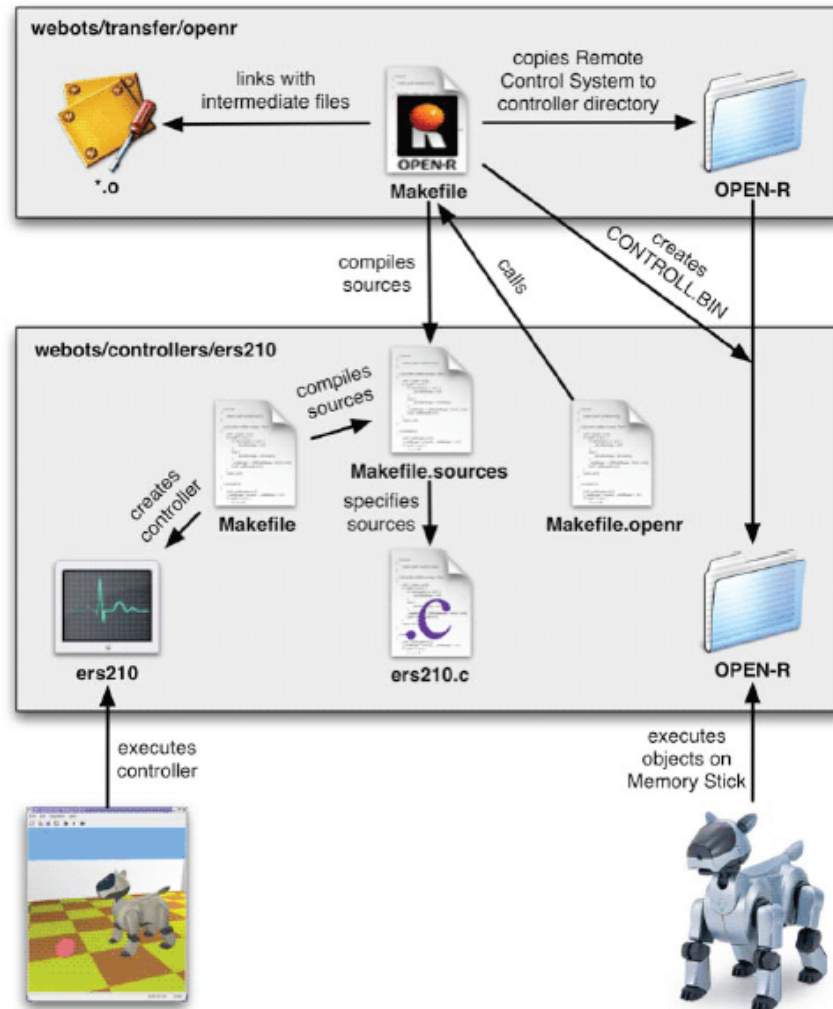


Figure 8.18: Aibo cross-compilation scheme

Cross-compiling your controller

For the sake of simplicity, we will describe the cross-compilation procedure on the default `ers210` controller for the Aibo ERS-210 which comes with Webots (see subsection 8.6.2). It is quite straight-forward to adapt the procedure described here to any other Aibo controller.

The default Aibo ERS-210 controller is located in Webots `controllers/ers210/` directory. The controller code is contained in the file `ers210.c`. There are two makefiles:

- `Makefile` is for the compilation of the controller for simulated execution;
- `Makefile.openr` creates a binary object executable on Aibo.

The source code files to be compiled by both makefiles are listed in `Makefile.sources`.

For cross-compilation, the `transfer/openr/` directory contains intermediate cross-compiled files of the Controller OPEN-R object. The `Makefile.openr` makefile compiles all source code files specified in `Makefile.sources (.c, .cc or .cpp)` using Sony's cross-compiler and links them with the already existing files. If there is not yet a directory called `OPEN-R/` in the controller directory, a default `OPEN-R/` directory is copied from `transfer/openr/`. Calling

```
$ make -f Makefile.openr clean
```

will again remove the `OPEN-R/` directory. The controller binary file `CONTROLL.BIN` resulting from the controller cross-compilation is then placed into the `OPEN-R/MW/OBJS/` subdirectory in the `controllers/ers210/` controller directory. The binary files of the other remote software objects are also located in that target subdirectory. (Initially, the `CONTROLL.BIN` binary code already in place corresponds to a *void* controller.) Four MTN motion sequence files are already present in `OPEN-R/MW/DATA/P/` and thus do not need to be uploaded separately.

Important: The files in `OPEN-R/` directory must not be modified, renamed, moved or deleted. This `OPEN-R/` directory may be directly copied to the Memory stick in order to install the complete remote software together with the cross-compiled controller. If you have already set up the remote software, as explained in section 8.2 (subsection 8.2.4 in particular), you need only copy the `OPEN-R/MW/OBJS/CONTROLL.BIN` controller binary file (to the corresponding location on the Memory stick), or use the **Cross** feature of the Control Panel (see subsection 8.5.1).

Using external data files in cross-compiled controllers

When cross-compiling a controller which reads or writes to external data files, such as MTN motion sequence definition files, it is a good idea to reference them by filename only (no absolute or relative paths), e.g.:

```
MTN* mtn = mtn_new("WWFWD.MTN");
```

instead of the usual:

```
MTN* mtn = mtn_new("../..data/mtn/ers210/WWFWD.MTN");
```

(Otherwise, Aibo will not be able to find the file.) To run such controller in simulation, e.g., a controller which uses MTN playback, you must have the external files in the same directory as your controller; you can either make a local copy of the file(s), or (if you're running Linux) simply create soft links. To run the controller on the real robot, you must of course also copy the required data files to the Memory stick.

Important: Remember: All Aibo data files are located in `/MS/OPEN-R/MW/DATA/P/` directory on the Memory stick. This is where you need to copy any MTN file your controller uses if you want to do it by hand; using the MTN **Upload** functionality of the Control Panel uploads the file automatically to the correct location.

Chapter 9

ALife Contest

A programming contest based on Webots was organized on the Internet. The web site of the contest¹ may provide more up to date information about it than this manual. ALife stands for "Artificial Life".

9.1 Previous Editions

This was actually the third edition of the ALife contest. Two editions were organized in 1999 and 2000. Each competition gathered about 10 teams worldwide made up of one to three individuals. The winners were respectively Keith Wiley from the University of New Mexico, USA and Richard Szabo from Budapest University, Hungary.

9.2 Rules

9.2.1 Subject

Two robots are roaming a maze-like environment (see figure 9.1), looking for energy. Energy is provided by chargers (see figure 9.2). However, chargers are scattered all around the environment and it is not so easy for the robots to find them. Moreover, once used by a robot, a charger will be unavailable for a while (see figure 9.3). Hence, the robot will have to go away and look for another charger. A robot will die if it fails finding an available charger before it runs out of energy. Then, the remaining robot will be declared the winner of the match.

The world configuration is chosen randomly for each match. A number of world configurations is provided within the Webots package, They are named `alife.wbt`, `alife1.wbt`, `alife2.wbt`, etc. Please note that the initial position and orientation of the robots may also be chosen randomly.

¹<http://www.cyberbotics.com/contest/>

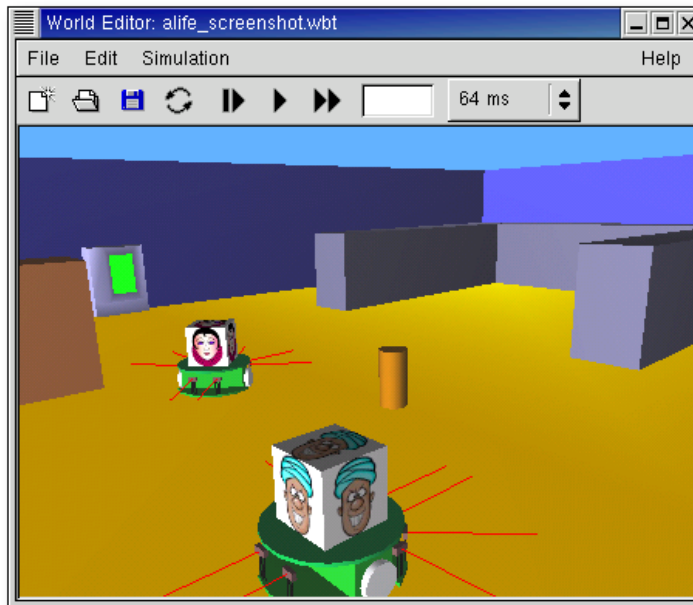


Figure 9.1: The world used in the contest

9.2.2 Robot Capabilities

All robots have the same capabilities. They are based on a model of Khepera robot equipped with a K6300 color matrix vision turret. Hence each robot has a differential wheels basis with incremental encoders, eight infra-red sensors for light and distance measurement, and a color matrix camera plugged on the top of the robot, looking in front. The resolution of this camera was scaled down to 80x60 pixels with a color depth of 32 bits. As you may have already understood, analyzing the camera image is a crucial issue in developing an efficient robot controller and you probably need to perform vision based navigation, using landmarks and mapping.

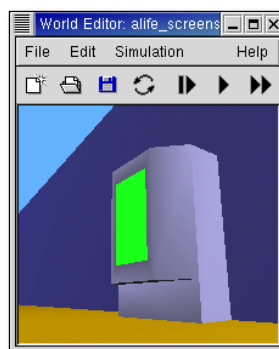


Figure 9.2: A charger full of energy

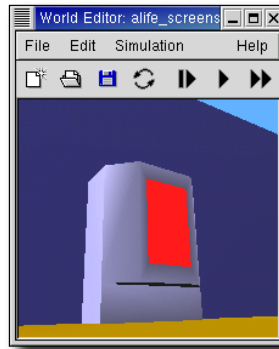


Figure 9.3: An empty charger

9.2.3 Programming Language

For the contest, the robots can be programmed in Java only. This ensures that the binaries carry no viruses or cheating systems. Hence, the executable files (`.class` files) can be easily shared among competitors without disclosing source code. Beware, that very good Java decompilers exist and that it may be possible for a cheating competitor to restore your code from your `.class`. He will just miss your comments... You may protect your Java code from such piracy by obfuscating it using a Java code obfuscator. This will make the code resulting from Java decompilation very difficult to understand, and practically unusable. Free even open source Java source code obfuscators may be found on the Internet.

There is no limit on the computation time a robot can use. However, since the simulator runs approximately in real time without any synchronization with the robots, robots performing extensive computations may miss some sensor information or react too late in some critical situations.

9.2.4 Scoring Rule

Once submitted on the web site, your robot will be appended at the bottom of the hall of fame. Then, it will engage matches each round. If n robots are present in the hall of fame, $n-1$ matches will be played each round. The first match will confront the last robot (bottom rank in the hall of fame) to the last but one robot (rank $n-1$). If the last robot wins, the two robots will swap their positions in the hall of fame, making the last robot win one position and the last but one robot fall down to the bottom position. Otherwise, nothing is changed. Then, the new last but one robot (which may have just changed) will play against the last but two robot. If the last but one robot wins, they will swap their positions, otherwise nothing occurs. And so on until we reach the top of the hall of fame. This way a robot can theoretically climb up from the bottom to the top position within a single round. However, a robot can lose only one rank per round. This is to encourage new competitors to submit their robots and have a chance to climb up the hall of fame rapidly. A round will be played every day during the contest.

It is always possible to introduce a new version of an existing robot controller, by simply uploading the versions of the `.class` files, erasing any previous ones. When a new version of a robot controller is introduced in the contest, its position in the hall of fame remains unchanged. The next matches are run using the new version.

9.2.5 Participation

The contest is open to any people from any country. Competitors may choose run for themselves or to represent their university or company. However, although competitors can update their robot controller by submitting new versions, only a single robot controller per competitor is allowed. If someone submits several robot controllers with different names into the contest, this person and the corresponding robot controllers will be banned out the contest.

9.2.6 Schedule

The contest started on July 1st 2002. From this date, competitors could download all the contest material and develop their robot controller. Matches between resulting controllers are held continuously from the middle of the summer until the end of the competition, on May 1st 2003. It is possible to enter the contest at any time before May 1st, 2003.

9.2.7 Prize

The winner of the contest will be the robot ranked at first position on May 1st, 2003. The authors of this robot will receive a Khepera II robot and a Webots PRO package (see figure 9.4).



Figure 9.4: First prize: a Khepera II robot and a Webots PRO package.

9.3 Web Site

The web site of the contest² allows you to view matches running in real time, to view the results, especially the hall of fame that contains the ranking of the best robots with their score. It is also possible to visit the home page of each robot engaged in the contest, including a small description of the robot's algorithm, the flag of the robot and possibly the e-mail of the author. You can even download the Java binary controller (`.class` files) of the some robots. This can be useful to understand why a robot performs so well and to confront on your computer your own robot against a possibly better one.

9.4 How to Enter the Contest

If you are willing to challenge the other competitors of the contest, here is the detailed procedure on how to enter the ALife contest. You will need either a Windows or a Linux machine to program your robot controller.

9.4.1 Obtaining the software

All the software for running the contest may be obtained free of charge.

- The Webots software to be used for the contest is available from the Webots download page³. This is an evaluation version of Webots which contains all the necessary material to develop a robot controller for the contest, except the Java environment. Follow the instructions on the Webots download page to install the Webots package.
- The Java 2 Standard Edition (J2SE) Software Development Kit (SDK) may be downloaded from Sun web site⁴ for free. Please use the version 1.4 of the SDK. Follow the instructions from Sun to install the SDK.

9.4.2 Running the software

Launch Webots and open the world named `alife.wbt`. Click on the **run** to start the simulation. You will see two robots moving around in the world. Each robot is controlled by a Java program named respectively `ALife0` and `ALife1` located in the Webots `controllers` directory. You may enter their directory and have a look at the source code of the programs.

²<http://www.cyberbotics.com/contest/>

³<http://www.cyberbotics.com/products/webots/download.html>

⁴<http://java.sun.com/j2se/1.4/download.html>

9.4.3 Creating your own robot controller

The simplest way to create your own robot controller is to start from the existing `ALife0` or `ALife1` controllers.

Installation

It is safer and cleaner to install a local copy of the material you will need to modify while developing your intelligent controller. Here is how to proceed:

1. Create a working directory which you will store all your developments. Let's call this directory `my_alife`. It may be in your Linux home directory or in your Windows `My Documents` directory or somewhere else.
2. Enter this directory and create two subdirectories called `controllers` and `worlds`.
3. Copy the file `alife.wbt` from the Webots `worlds` directory to your own `worlds` you just created. Copy also the `alife` directory and all its contents from the Webots `worlds` directory to your own `worlds` directory. You may replace the images `ALife0.png` and `ALife1.png` in the `alife` directory by your own custom images. These images are actually texture flags associated to the robots. Their size must be 64x64 pixels with 24 or 32 bits depth. They should not represent a green rectangle, possibly faking the face of a charger and hence confusing the opponent. If a flag appears to be a charger fake, it will be removed.
4. Copy the whole `ALife0` directory from the Webots `controllers` directory to your own `controllers` directory you just created. Repeat this with the `ALife1` directory. This way you could modify the example controllers without losing the original files.
5. In order to indicate Webots where the files are, launch Webots, go to the **File** menu and select the **Preferences...** menu item to open the Preferences window. Select the **Files and paths** tab. Set `alife.wbt` as the Default world and indicate the absolute path to your `my_alife` directory, which may be `/home/myname/my_alife` on Linux or `C:\My Documents\my_alife` on Windows.

From there, you can modify the source code of the controllers in your `controllers` directory, recompile them and test them with Webots.

Modifying and Compiling your controller

If you know a little bit of Java, it won't be difficult to understand the source code of the `ALife0` and `ALife1` controllers, which are stored respectively in the `ALife0.java` and `ALife1.java`. You may use any standard Java objects provided with the Java SDK. The documentation for

the `Controller` class is actually the same as for the C programming interface, since all the methods of the `Controller` class are similar to the C functions of the Controller API described in the Webots Reference Manual, except for one function, `robot.live` which is useless in Java. Before modifying a controller, it is recommended to try to compile the copy of the original controllers.

To compile the `ALife0` controller, just go to the `ALife0` directory and type the following on the command line:

```
javac -classpath "C:\Program Files\Webots\lib\Controller.jar;." ALife0.java
on Windows.
```

```
javac -classpath "/usr/local/webots/lib/Controller.jar:." ALife0.java
on Linux.
```

If everything goes well, it should produce a new `ALife0.class` file that will be used by Webots next time you launch it (or reload the `alife.wbt` world).

Now, you can start developing! Edit the `ALife0.java`, add lines of code, methods, objects. You may also create other files for other objects that will be used by the `ALife0` class. Test your controller in Webots to see if it performs well and improve it as long as you think it is necessary.

9.4.4 Submitting your controller to the ALife contest

Once you think you have a good, working controller for your robot, you can submit it to the online contest. In order to proceed, you will have to find a name for your robot. Let's say "MyBot" (but please, choose another name). Copy your `ALife0.java` to a file named `MyBot.java`. Edit this new file and replace the line:

```
public abstract class ALife0 extends Controller {
```

by:

```
public abstract class MyBot extends Controller {
```

Save the modified file and compile it using a similar command line as seen previously. You should get a `MyBot.class` file that you could not test, but that will behave the same way as `ALife0.class`.

Register to the contest from the main contest web page⁵, providing "MyBot" as the name of the robot. Then, upload all the necessary files in your `MyBot` directory. This includes the following:

- `MyBot.class` file and possibly some other `.class` files corresponding to other java objects you created (it is useless to upload the `ALife0.class` file)

⁵<http://www.cyberbotics.com/contest>

- A text file named `description.txt` of about 10 lines that may include some HTML tags, like hyperlinks.
- A PNG image named `flag.png` that will be used as a texture to decorate your robot, so that you can recognize it from the webcam. This image should be a 64x64 pixels with a bit depth of 24 or 32. It should not represent a green rectangle, trying to fake the face of a charger, otherwise it will be cancelled.

That's it. Once this material uploaded, your robot will automatically enter the competition with an initial score of 10. A contest supervisor program will use you controller to run matches and update your score and position in the hall of fame. You can check regularly the contest web site to see how your robot performs.

9.4.5 Analysing the performance and improving your competing controller

Match movies

During each round, a number of match movies are generated and stored in the `results` directory of the contest home page. These files can be played back with Webots. Just download them and save them in the `alife_playback` directory which lies in the Webots `controllers` directory. Rename the file to `match.dat` (overwriting the existing `match.dat` file) and open the world named `alife_playback.wbt` with Webots. You should then see the match playback running. To know who was the winner in a `.dat` file, just look at the two bottom lines of the file. If the last line ends with 0, then the first robot wins (i.e., its name is displayed on the first line of the file). Otherwise the second robot wins.

Debug and error log

In order to debug your program, or at least to understand what went wrong or right during a round match, you can save data into a log file. This will help you developing your controller, especially on Windows where the DOS console closes immediately after a controller crashes and doesn't let you read the printed messages in this console. Moreover, it may also be useful to do it during the contest matches running on the match server to understand exactly how your controller behaved during a contest match. Your log file can be retrieved from the match server after the round completed as a zipped file.

To proceed, you first need to create such a log file and then log useful information using the `println` statement:

```
import java.io.*;
...
PrintStream log;
FileOutputStream file;
```

```

try {
    file = new FileOutputStream("log.txt");
    log = new PrintStream(file);
} catch (Exception e) { }
...
log.println("My estimated coords: (" +x+", "+y+") my state="+state);
...
log.println("My energy level: "+energy);
...
log.close();
file.close();
...

```

During each round, for each competitor using this log file facility, a log file called `log.zip` is stored in the controller directory of the `competitors` directory of the contest home page. This file is the compressed version of your `log.txt` file. It contains all the debug messages produced by your controller along the different matches of the last round. Please note that this log file will be visible by all the other competitors, so be cautious and don't reveal your secret algorithms. Also useful, in the `results` directory, a file called `errors.zip` contains the error log of the last round, which may be useful to detect if your controller crashed, producing a java exception. Note that these files are erased at the beginning of each new round and replaced by new ones corresponding to the new round.

Robot memory

It may be useful for your robot to store some data corresponding to knowledge acquired across the different matches. Such data should be saved regularly during a normal run or, if you prefer, just when the controller energy reaches a small value (like below 3), that is the match is about to complete. The data can be in turn re-read by the controller when it starts up a new match, to refresh its memory. Here is how to implement it:

```

import java.io.*;

// to create/write into the file
Random r = new Random();
try {
    DataOutputStream s;
    s=new DataOutputStream(new FileOutputStream("memory.dat"));
    s.writeInt(100); // save 100 int
    for (int i=0; i<100;i++) s.writeInt(r.nextInt(100));
    // you should rather save some useful info
    // here instead of random garbage!
} catch (Exception e) {

```

```
e.printStackTrace(System.out);
}

// to read from that file
try {
    DataInputStream s =
    new DataInputStream(new FileInputStream("memory.dat"));
    int t = s.readInt(); // read the size of the data
    int[] a = new int[t];
    for(int i=0; i<t; i++) a[i] = s.readInt(); // read back my garbage
    for (int i=0; i<t; i++) System.out.print(a[i]+"\\t" );
} catch (Exception e) {
    e.printStackTrace(System.out);
}
```

The `memory.dat` file of each competitor is also made available for download to all competitors on the contest web site. This file is stored at the same place as the `log.zip` file, that is, within the controller directory of the `competitors` directory on the contest web site.

9.5 Developers' Tips and Tricks

This section contains some hints to develop efficiently an intelligent robot controller.

9.5.1 Practical issues

The `ALife0` example program display a Java image for showing the viewpoint of the camera, after some image processing. This is pretty computer expensive and you may speed up the simulation by disabling this display, which should be used only for debug. By the way, during contest matches, the Java security manager is set so that your Java controller cannot open a window or display anything.

9.5.2 Java Security Manager

To avoid cheating or viruses, a Java security manager is used for contest matches ran by the automatic contest supervisor. This security manager will prevent your Java controller from opening any file for writing or reading and doing any networking stuff.

9.5.3 Levels of Intelligence

It is possible to distinguish a number of level in the complexity of the control algorithms. These level can be ranked as follow:

1. The robot is able to move and avoid obstacles. However, it does not use the camera information at all and will find chargers only by chance. This correspond to the `ALife0` controller.
2. In addition to level 1, the robot is able to recognize if a full charger is in front of it, even far away. In this case, it will be able to adjust its movement to reach the charger if not obstacles are on the way. Otherwise, the robot will look into another direction for chargers.
3. In addition to level 2, the robot is able to move around obstacles preventing a movement toward a full charger.
4. In addition to level 3, the robot is able to perform an almost complete exploration of the world, reaching places difficult to reach for simpler robots (you will rapidly notice that some places are more difficult to reach than others, the problem is that these places may contain chargers...).
5. In addition to level 4, the robot is able to build a map of its environment (mapping), so that once a charger is found, it is placed on the map, thus facilitating the procedure for finding it back. After completing the map, the robot can efficiently navigate between chargers without loosing time to search for them.
6. In addition to level 5, the robot tries to chase its opponent, blocking it, preventing it to reach chargers or emptying chargers just before it arrives.

During the previous editions of the contest, the best competitors reached level 4 (and even one reached level 5 after the contest ended). We believe that reaching level 5 or 6 may lead to significant performance improvements and probably to the first place of the hall of fame...

Chapter 10

Robot Soccer Lab

Robotics soccer has become an increasingly attractive research application for mobile robotics. Many contests are organized world wide, among them the most famous are probably the FIRA contest and the RoboCup contest. This chapter will get you started with a robot soccer application in Webots.

10.1 Setup

Webots contains a setup for robotics soccer as depicted in figure 10.1 . This setup is freely inspired from the official FIRA Small League MiroSot Games Rules. It can be modified to suit your needs.



Figure 10.1: A soccer simulation in Webots: soccer.wbt

Each team is composed of three robots. Each robot has a controller program which is aware of the the position and orientation of every robot in the soccer field. Each robot can drive its motors wheels to move in the soccer field. A supervisor process is responsible for counting the time. By default, a match lasts for 10 simulated minutes which may correspond to 1 minute if your computer is powerful and if you run the match without the real time option checked in. The supervisor process also counts the goals and reset the ball and the robots to their initial positions after a goal has been scored.

10.2 Rules

The rules are very simple: you have to drive your robots so that you score a maximum of goal within the 10 minutes of the match. There are no fouls, no penalty kick or free kick.

There is no obligation to have a goal keeper, you may decide to have three players all over the field, or to have one, two or even three goal keepers!

You cannot modify robots, i.e., change their shape, add sensors, etc.

10.3 Programming

In order to program your robot, a single controller program is used for each team. The `soccer_blue` controller program is used for the blue team while the `soccer_yellow` controller program is used for the yellow team. Each of these controller programs will be run as three concurrent processes. In each instance of these programs, a test is done to determine the number of the robot which can be 1, 2 or 3, according to the name of the `DifferentialWheels` node. One can also test the team color the same way. The provided examples shows how to distinguish the goal keeper (number 3) from the other players (numbers 1 and 2). Hence, it is possible to have a generic `soccer.c` source code and to compile it to either a `soccer_blue.exe` or a `soccer_yellow.exe` executable file. Please note that on Linux and Mac OS X, the `.exe` extension is not used.

In order to get starting programming a robot soccer team, you should have a look in details to the `soccer_blue.c` or `soccer_yellow.c` source codes. These examples shows how to obtain the `x`, `y` and orientation for each robot from the supervisor, as well as the coordinates of the ball. They contain useful macros for that. Moreover, they show how to program each independant robot according to its number. Finally, they show how to make a fairly intelligent goal keeper that will get placed according to the ball position. The behavior of players 1 and 2 is random in this example and it is up to you to make them more intelligent!

10.4 Extensions

This very simple robotics soccer system can be configured or extended according to your needs.

10.4.1 Modifying the soccer field

It is possible to redesign the soccer field as you need. You can enlarge it, resize the goals, change the ground texture, etc. Moreover, you can change the ball properties, like its mass, its bounce parameter, etc. All these changes are possible from the scene tree window. For resizing the field, you will have to edit the coordinates of the components of the field. It will also be necessary to update the respective bounding objects accordingly.

For example, if you want to change the bounce parameter of the ball to make it bounce less, just double click on the ball, open the ball node in the scene tree window, open the physics node of the ball node and set the bounce parameter to 0.2 instead of 0.7. This will make the ball.

10.4.2 Modifying the robots

Similarly, it is possible to modify the robots. You can change the number of robot per team, add new sensors to the robots, like distance sensors or cameras, remove the receiver sensor if you want to prevent the robots to be aware of global coordinates provided by the supervisor. All these operation can be performed through the scene tree window, using copy and paste functions and editing the robots properties. This way, it is possible to turn the soccer robots into fully autonomous robots relying only on local information and not on global coordinates provided by the supervisor.

10.4.3 Modifying the match supervisor

If you would like to modify the rules, you will probably have to modify the match supervisor. This is a small C supervisor controller program called `soccer_supervisor` lying in the `controllers` directory. The match supervisor has only three functions: (1) it measures the time decreasing from 10 minutes to zero, (2) it count the goals, update the score and reset the robots and the ball after a goal and (3) it provides each robot with global coordinates and orientation for each robot and global coordinates for the ball. You may change any of these features, and add additional features, like fouls when a robot hits another robot.

For example, let's assume you want that the robots should not touch each other, otherwise a penalty kick is called. Your supervisor program should compute the distance between each robots of different teams. If this distance drops below the size of a robot, you call the penalty. Do to so, just set the ball and robots positions so that the robot which benefit of the penalty kick is ready to kick.

This way, it is possible to add many new rules, like prevent the goal keeper to leave its goal, etc.

