Webots User Guide

release 3.2.9

www.cyberbotics.com

July 10, 2002

Trademark information

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 programming and in VRML 2.0 (Virtual Reality Modeling Language).

The great innovation of the third version of Webots is that any robot with two-wheel differential steering can be modelled and simulated. Predefined objects like shapes, sensors and axles allow users to create and run their own simulated robot. Thus, Webots is no longer limited to the Khepera and Alice robots.

The GUI of Webots version 2 has been replaced by GTK+, an Open Source Free Software GUI toolkit.

If you have already developed programs using Webots 2.0, please read chapter 2 to update your programs to run with the new version.

We hope that you will enjoy working with Webots .

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 Jordi Porta, Emanuele Ornella, Yuri Lopez de Meneses, Auke-Jan Ijspeert, Gerald Foliot, Allen Johnson, Michael Kertesz, Aude Billiard, 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-read this guide.

Contents

Chapter 1

Installing Webots

1.1 Hardware requirements

Webots is available for RedHat 7.2 Linux i386, Windows 95, Windows 98, Windows ME, Windows NT, Windows 200 and Windows XP. Other versions of Webots for other UNIX systems (MacOS X, Solaris, Linux PPC, Irix) may be available upon request.

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

1.2 Registration Procedure

1.2.1 Webots license

Starting with Webots , 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 and user ID. This encrypted file will enable you to use Webots according to the license you purchased. This file is stricly 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.

1.2.2 Registering

In order to register your copy of Webots and get the license file, you will have to fill out a form[1] on the website of Cyberbotics (see figure 1.1). You will then receive an e-mail containing the `webots.key` file corresponding to your license.

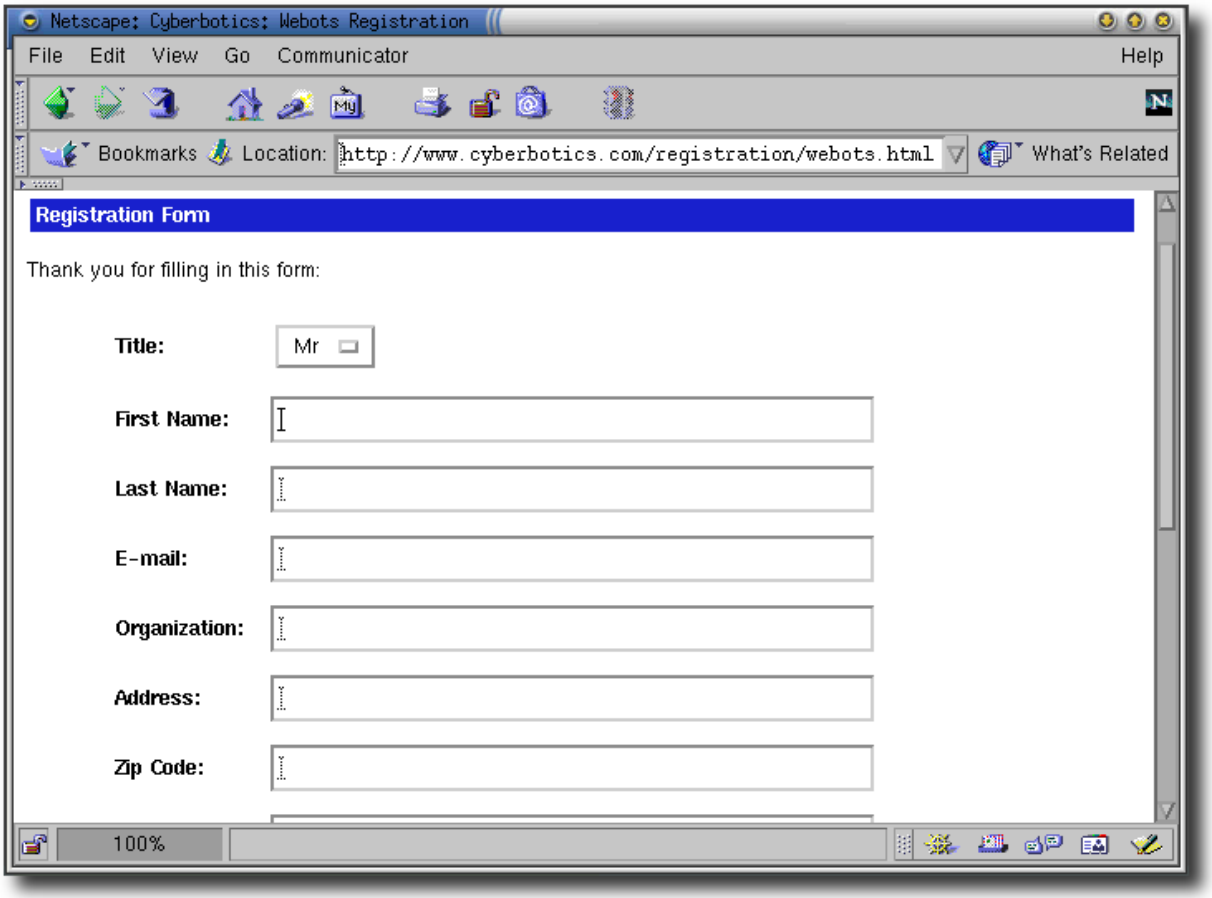


Figure 1.1: Webots registration page

Please take care to properly fill in each field of this form. The *Serial Number* is the serial number of your Webots package which is printed on the back side of the CD-ROM package under the heading *S/N:*.

[1] `http://www.cyberbotics.com/registration/webots.html`

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.

1.3 Installation procedure

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

1.3.1 RedHat 7.2 Linux i386

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
   rpm -Uvh /mnt/cdrom/linux/lib/miniGLU-3.0-1.i386.rpm
   rpm -Uvh /mnt/cdrom/linux/lib/solid-2.1.0-1.i386.rpm
   rpm -Uvh /mnt/cdrom/linux/lib/gnet-1.1.0-1.i386.rpm
   rpm -Uvh /mnt/cdrom/linux/lib/gtkglarea-1.2.2-2.i386.rpm
   rpm -Uvh /mnt/cdrom/linux/lib/mpeg_encode-1.5b-4.i386.rpm
   # mpeg_encode is useful only if you want to export MPEG movies
   rpm -Uvh /mnt/cdrom/linux/webots/webots-3.2.9-1.i386.rpm
   rpm -Uvh /mnt/cdrom/linux/webots/webots-kros-1.0.1-1.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 the `rpm` fails to install the packages.
3. Copy your personal `webots.key` file into the `/usr/local/webots/server/resources` directory where Webots was just installed.

1.3.2 Windows 95, 98, ME, NT, 2000 and XP

1. Uninstall any previous release of Webots, if any, from the **Start** menu, **Control Panel**, **Add / Remove Programs**. or from the **Start** menu, **Cyberbotics**, **Unistall Webots**.
2. Insert the Webots CD-ROM and open it.
3. Go to the `windows` directory on the CD-ROM.

4. Double click on the `WEBOTS-3.2.9_SETUP.EXE`.
5. Follow the installation instructions.
6. Copy your personal `webots.key` file into the `C:\Program Files\Webots\resources` directory where Webots was just installed.

In order to be able to compile controllers, you will need to install a C/C++ development environment. We recommand to use Dev-C++ which is provided on the Webots CD-ROM (in the `windows/utils` directory) as well as from the Bloodshed.net[2] web site. Dev-C++ is an integrated development environment (IDE) for C/C++ with syntax highlighing 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++™ if you own a license of this software.

[2] `http://www.bloodshed.net`

Chapter 2

Upgrading from Webots 2

If you have already worked with Webots 2, your existing programs need to be modified for use with Webots .

2.1 World

You first have to edit your Webots 2.0 world with a text editor to make it understandable by Webots . This operation is however fairly straightforward as explained in the following instructions:

2.1.1 Header of the file

The first line of the file has to be changed as follows:

Replace

```
#WEBOTS V2.0 utf8
```

by

```
#VRML_SIM V3.0 utf8
```

After this header, remove

```
Group { children [
```

at the beginning of the file and

```
] }
```

at the end of the file.

2.1.2 Nodes

Some Webots 2 nodes no longer exist: `Ball`, `Can`, `Ground`, `KheperaFeeder`, `Lamp`, `Wall`, `Alice`, `AliceIRCom`, `Khepera`, `KheperaK213`, `KheperaK6300`, `KheperaGripper`, `KheperaPanoramic`. They must be replaced by their new equivalents, as illustrated in table 2.1.

Nodes in Webots 2	Nodes in Webots	including (Webots)
Ball	Solid	Sphere
Can	Solid	Cylinder
Ground	Solid	ElevationGrid
Lamp	PointLight	
Wall	Solid	Extrusion
Alice	DifferentialWheels	DistanceSensor
AliceIRCom	Receiver	
Khepera	DifferentialWheels	DistanceSensor
KheperaFeeder	Charger	
KheperaGripper	Servo	Gripper
KheperaK213	Camera	
KheperaK6300	Camera	
KheperaPanoramic	several Camera	

Table 2.1: Node equivalents between Webots 2 and Webots

2.2 Controller

2.2.1 Location

The controller program is still found in the `controllers` directory of your user directory defined in the Webots preferences. However, the name of the directory for each controller has changed: rename `yourcontroller.khepera` to `yourcontroller` (simply remove the `.khepera` extension). The same applies for the alice and supervisor controller directories where the `.alice` and `.supervisor` extensions must be removed. Note that if you used the same prefix for both khepera and a supervisor controller (e.g. `stick_pulling.khepera` and `stick_pulling.supervisory`), you will have to rename one of them because you cannot have two directories with the same name. For example, the `stick_pulling.khepera` directory can be renamed to `stick_pulling` and the `stick_pulling.supervisor` directory can be renamed to `stick_pulling_supervisor`.

2.2.2 Khepera

The khepera_xxx functions have disappeared. You must replace them with their counterparts of the new API as illustrated on table 2.2:

Webots 2	Webots
khepera_live	robot_live
khepera_die	robot_die
khepera_step	robot_step
khepera_set_speed	differential_wheels_set_speed
khepera_enable_proximity	distance_sensor_enable
khepera_disable_proximity	distance_sensor_disable
khepera_get_proximity	distance_sensor_get_value

Table 2.2: Some equivalent function calls between Webots 2 and Webots

Morover, the #include <Khepera.h> must be replaced by the following:

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

2.2.3 Alice

The alice_xxx functions have also disappeared. Now you can program the Alice robot (indeed a differentially wheeled robot) just like any other robot in Webots. Thus all the functions described in the above subsection also apply to an Alice robot model.

2.2.4 GUI

The gui_xxx functions have all disappeared and should now be replaced by GTK+ functions. GTK+ is much more powerful than the GUI provided with Webots 2 and is well documented. You can find the GTK+ documentation (Tutorial and Reference Manual) on the Webots CD-ROM, in books available in computer science libraries, and on the GTK+ web site[1].

2.2.5 Supervisor

The syntax of supervisor functions has changed a lot from Webots 2 to be more consistent with the rest of the API. For example, a supervisor is now considered as a robot, hence the supervisor_step function has been replaced by the robot_step function. However,the way in which the supervisor interacts with webots remains unchanged.

[1] http://www.gtk.org

Chapter 3

Getting Started with Webots

To run a simulation in Webots, you need two 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 later on in this book.

3.1 Launching Webots

3.1.1 On Linux

From an X terminal, type `webots` to launch the simulator. You should see the world window appear on the screen (see figure 3.1).

3.1.2 On Windows

From the **Start** menu, go to the **Program Files — Cyberbotics** menu and click on the **Webots 3.2.9** menu item. You should see the world window appear on the screen (see figure 3.1).

3.2 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.

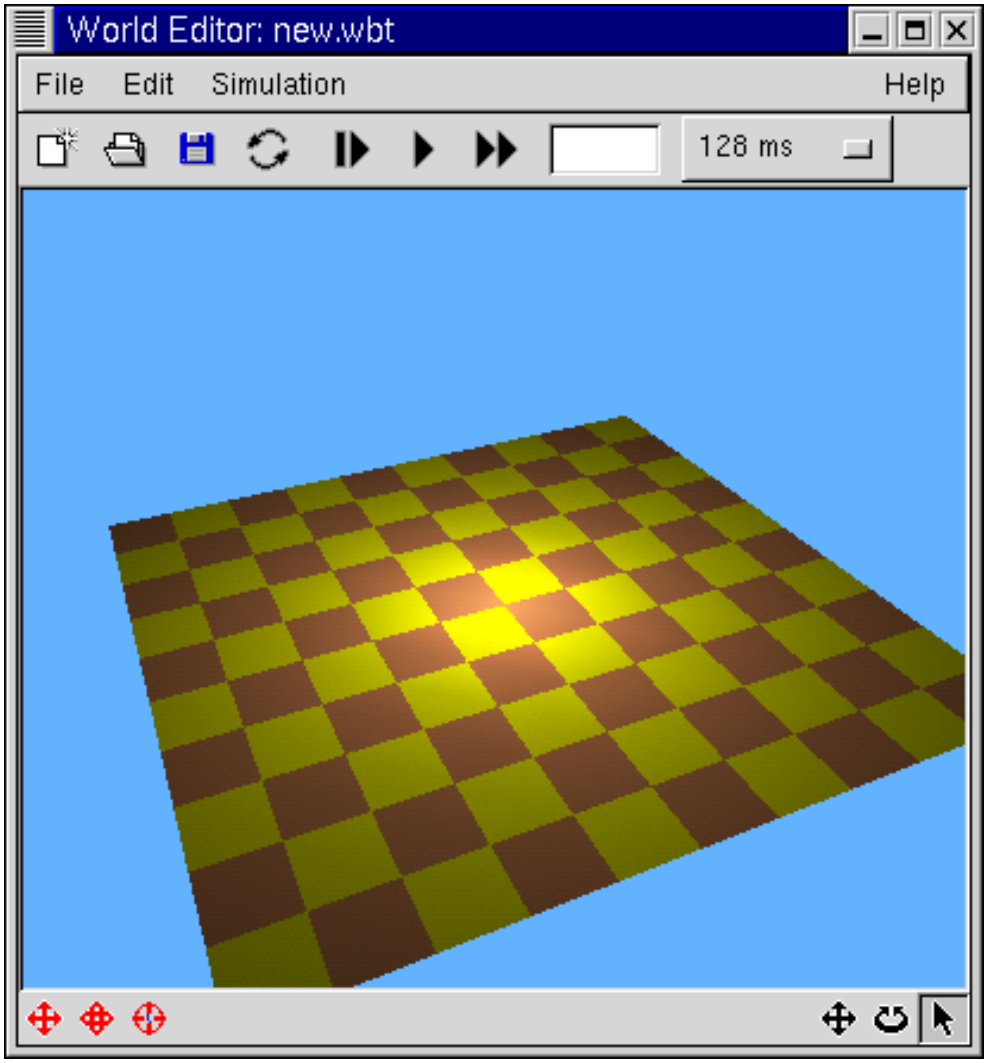


Figure 3.1: Webots main window

3.2.1 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:

 **New**

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:

 **Open...**

 **Save**

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

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:

 **Revert**

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).

  The `Refresh rate` corresponds to the speed of the simulation when Webots is launched.

  The `HyperGate port` specifies the computer port used for HyperGate networking.

- **Rendering**: this tab controls the 3D rendering in the simulation window.

  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 default `.wbt` world which is open when launching Webots and the user directory are defined here. The user directory should contain at least a `worlds`, `controllers`, and `objects` directories where Webots will be looking for files.

3.2.2 Edit menu

The **Scene Tree Window** 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 (see subsubsection 3.3.3).

3.2.3 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 simulation step.

 **Run**: execute simulation steps until the **Stop** mode is entered.

 **Fast**: same as **Run**, except that no display is performed.

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 stays blank until the **Fast** mode is stopped.

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 3.2) allows you to observe the speed of the simulation on your computer. It 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 fast as the corresponding real robots would. This information is relevant both in **Run** mode and **Fast** mode.

Figure 3.2: Speedometer

The time step can be chosen from the popup menu situated next to the speedometer. It indicates how frequently the display is refreshed. It is expressed in virtual time milliseconds. The value of this time step also defines the duration of the time step executed during the **Step** mode. The time step can be chosen from the popup menu situated next to the speedometer. It indicates how frequently the display is refreshed. It is expressed in virtual time milliseconds. The value of this time step also defines the duration of the time step executed during the **Step** mode.

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 without hardware acceleration and 4 on an Ultra Sparc 10 Creator 3D.

3.2.4 **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.

3.2.5 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 click on the left mouse button in the scene and then:

  if you click on a solid object and drag the mouse in the scene, the rotation will be centered around the origin of the local coordinate system of this solid object.

  if you click outside of any solids and drag the mouse the rotation will be centered around the origin of the world coordinate system.

- *Translate viewpoint*: To translate the camera in the *x* and *x* directions, you can click the right mouse button and drag the mouse on the scene.

- *Zoom / Tilt viewpoint*: set the mouse over the scene, then:

  if you click on the middle button and drag the mouse vertically, the camera will zoom in or out.

  if you click on the middle button and drag the mouse horizontally, the camera will rotate around its *z* axis.

  if you use the wheel of the mouse, the camera will zoon in or out.

  if you have a two-button mouse, hold down the control key while pressing the left mouse button to be able to zoom in the scene.

3.2.6 Moving a solid object

In order to move 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 click-and-drag rotates the selected object around its y axis.

- Lift: Pressing 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. If you have a two-button mouse, you will need to hold down both the shift and control keys to lift solid objects.

3.2.7 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.

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

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

3.3.1 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:

 **Cut**

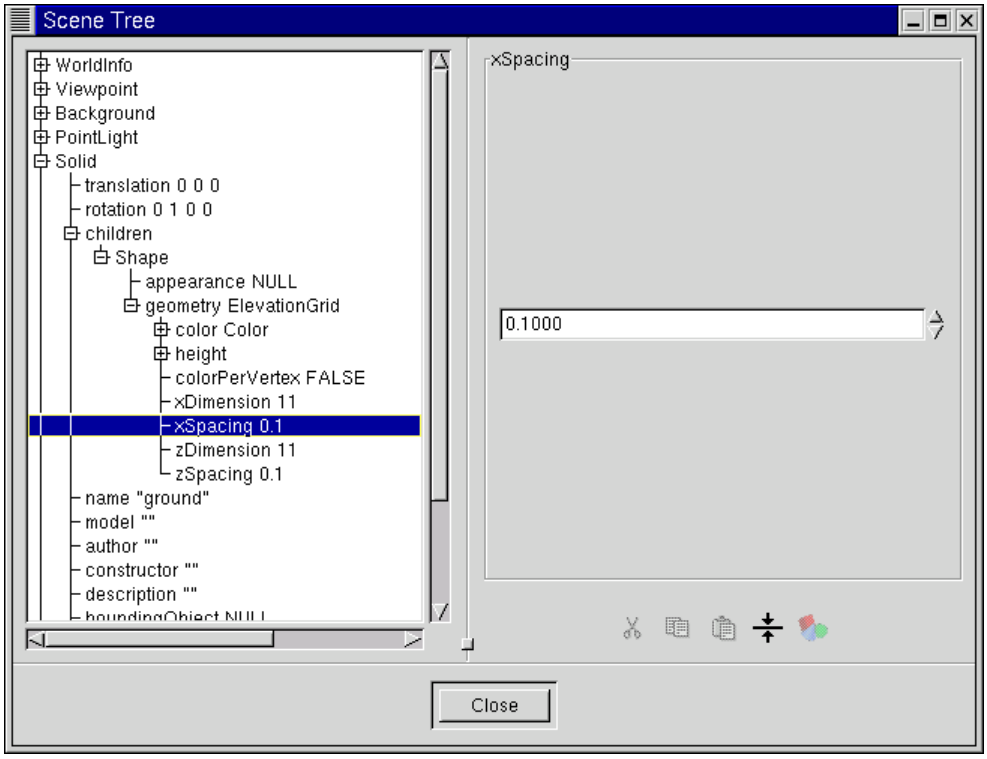


Figure 3.3: Scene Tree Window

 **Copy**

 **Paste after**

These three buttons let you cut, copy and paste nodes and fields. However, you can't perform these operations on the three first nodes of the tree (`WorldInfo`, `Viewpoint` and `Background`). These nodes are mandatory and don't need to 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 the default value(s) of a field. 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.

3.3.2 VRML nodes

A number of VRML 2.0 nodes are partially or completely supported in Webots.

The exact features of VRML 2.0 are the subject of a standard managed by the International Standards Organization (ISO/IEC 14772-1:1997).

You can find the complete specifications on the official VRML Web site: \texttt{http://www.vrml.org}.

The VRML nodes supported in Webots are the following:

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

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

The Reference Manual gives a more comprehensive list of nodes with associated fields.

3.3.3 Webots specific nodes

In order to implement powerful simulations including mobile robots with two-wheel differential steerings, a number of nodes specific to Webots have been added to the VRML set of nodes.

VRML 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 VRML `Transform` node, principally the `Solid` node. Other Webots nodes (`DifferentialWheels`, `DistanceSensor`, `Camera`, etc.) inherit from this `Solid` node.

The different fields of the Webots nodes are explained below.

The Reference Manual gives a complete list of Webots nodes and their associated fields along with a brief description of each field.

The Solid node

A solid is a group of shapes that you can drag and drop in the world, using the mouse. Moreover, the sensors of the robots and the collision detector of the simulator are able to detect solids. The `Solid` node represents this group of shapes in the scene tree.

Principle of the collision detection of the simulator:

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. list of `children` of the `Solid` node are used to compute the bounding object.

The collision detection is mainly of use between a robot `DifferentialWheels` node) and an obstacle (`Solid` node), and between two robots. Two `Solid` nodes can never interpenetrate each other; their movement is stopped just before the collision.

A description of the fields of the `Solid` node is given below.

The `Solid` node inherits from the VRML `Transform` node. The additional fields are:

- `name`: individual name of the solid (e.g.: ””my blue chair”).
- `model`: generic name of the solid (e.g.: ”chair”).
- `author`: name of the author of the simulation model of the solid.
- `constructor`: name of the company or individual who made the real solid.
- `description`: short description (1 line) of the solid.
- `boundingObject`: shape or a group of shapes which bound the solid for collision detection. If the value of this field is `NULL`, the collision detection is computed from the children list of the `Solid` node. In any other case, the `Shape` node is not used, because a bounding box is not a filled object. The `Shape` node is replaced directly by a `Box` or a `Cylinder` node for example. See the example below.
- `physics`: this field is used when it is necessary to model a minimum of physics for a `Solid` object. In this case, it contains a `Physics` object which defines a number of physical properties for the solid. This is especially useful when implementing a robot pushing an object like a ball. In this case, both the robot and the ball should have a `Physics` node in their `physics` field. If not used, this field is left to NULL.
- `joint`: unused at the moment (reserved for future use).
- `locked`: if TRUE, the solid object cannot be moved using the mouse. This is useful to prevent moving an object by error.

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 3.4.

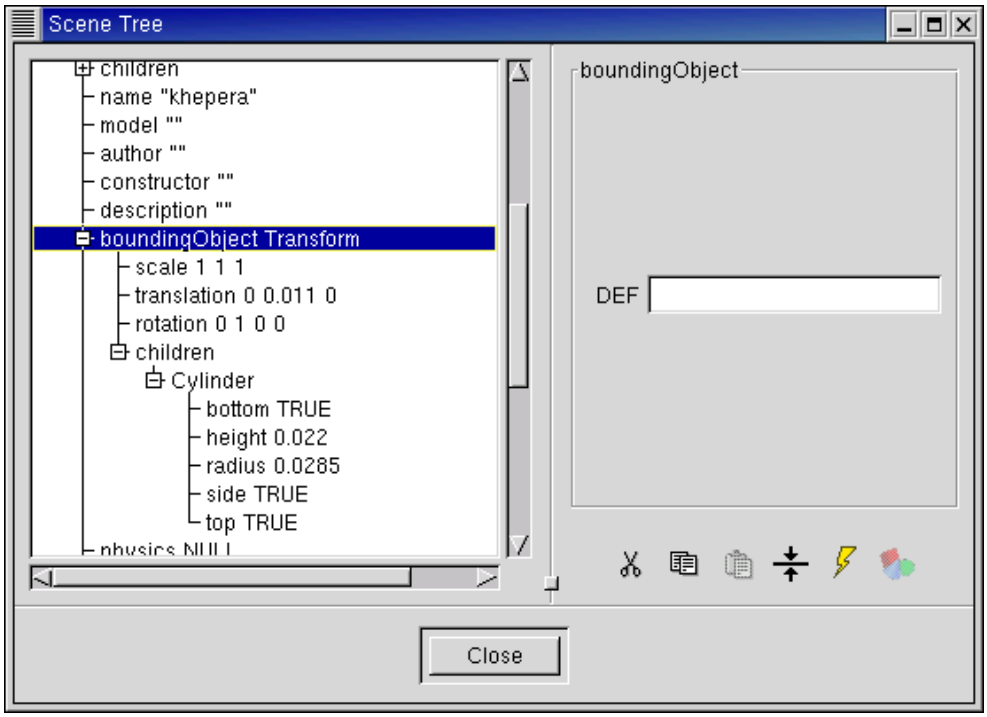


Figure 3.4: The bounding box of the Khepera robot

The DifferentialWheels node

The `DifferentialWheels` node inherits from the `Solid` node. It is used to represent any robot with two-wheel differential steering. The two specific fields which are essential for the simulation are `axleLength` and `wheelRadius`. The value of `axleLength` is the distance (in meters) between the two wheels of the robot, and the value of `wheelRadius` is the radius (in meters) of the wheels.

Moreover, the origin of the robot coordinate system is the projection on the ground plane of the center of the axle of the wheels. *x* is the axis of the wheel axle, *y* is the vertical axis and *z* is the axis pointing towards the rear of the robot (the front of the robot has negative *z* coordinates).

The `DifferentialWheels` node inherits from the `Solid` node. The additional fields are:

- `controller`: name of the program controlling the robot. This program lies in the directory with the same name in the controllers directory; for example, the `void` (or `void.exe`)

controller is found in the `webots/controllers/void/` directory . The simulator will use this program to control the robot.

- `synchronozation`: if the value is `TRUE` (default value), the simulator is synchronized with the controller; if the value is `FALSE`, the simulator runs as fast as possible, without synchronization.
- `battery`: this field should contain three values: the first one corresponds to the current energy of the robot in Joules(*J*), the second one is the maximum energy the robot can hold in Joules, the third one is the speed of energy recharge in Watts (*[W]=[J]/[s]*). The simulator updates the first value, while the two others remain constant.
- `cpuConsumption`: consumption of the CPU (central processing unit) of the robot in Watts.
- `motorConsumption`: consumption of the the motor in Watts.
- `axleLength`: distance between the two wheels in meters.
- `wheelRadius`: radius of the wheels in meters. Both wheels must have the same radius.
- `maxSpeed`: maximum speed of the wheels, expressed in *rad/s*.
- `maxAcceleration`: maximum acceleration of the wheels, expressed in *rad/s2̂*.
- `speedUnit`: defines the unit used in the `differential_wheels_set_speed` function, expressed in *rad/s*.
- `slipNoise`: slip noise added to each move expressed in percent. If the value is 0.1, a noise of +/- 10 percent is added to the command for each simulation step.
- `encoderNoise`: noise added to the incremental encoder. If the value is -1, the encoders are not simulated. If the value is 0, encoders are simulated without noise. Otherwise a noise is added to encoder values. When the robot faces an obstacle, the robot wheels do not slip, hence the encoder values are not incremented. This is very useful to detect that a robot has hit an obstacle.
- `encoderResolution`: defines the number of encoder incrementations per radian of the wheel. An `encoderResolution` of *100* will make the encoders increment their value of about *628* each times the wheel makes a complete revolution.

The DistanceSensor node

The `DistanceSensor` node is used to model sonar sensors, infra-red sensors and laser range finders. It uses a ray casting algorithm to detect collision between the sensor ray and `Solid` nodes in the world. The `DistanceSensor` node inherits from the `Solid` node. it includes two additional specific fields:

- `type`: type of sensor: currently only the ”infra-red” type is supported, but upcoming versions of Webots may include ”sonar” or ”laser” types. Infra red sensors have a special property: the are color sensitive and will see better light or red obstacles than dark or black ones.

- `lookupTable`: This field is best explained through an example: Let us consider an infra-red sensor. The white noise on the return value is 10 percent. For an obstacle made of a given material and color and for a given ambient light, the response of the sensor is as shown in figure 3.5

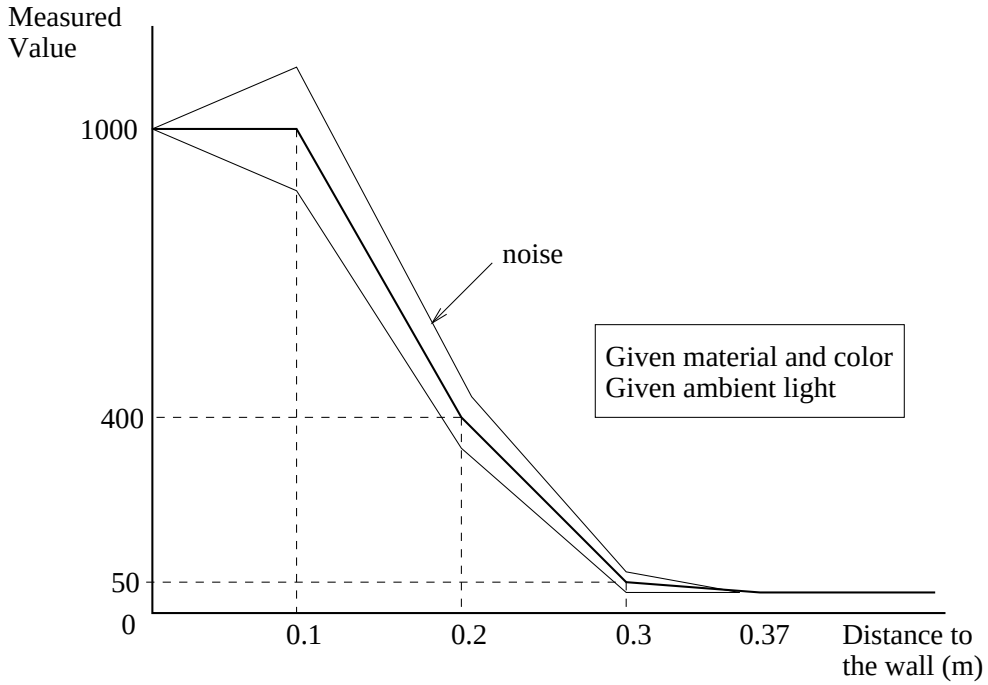


Figure 3.5: Measurements of the light reflected by an obstacle

The values of the `lookupTable` will be:

```
lookupTable [ 0     1000  0,
              0.1   1000  0.1,
              0.2    400  0.1,
              0.3     50  0.1,
              0.37    30  0   ]
```

This means that for a distance of 0 meter, the sensor will return a value of 1000 without noise (0), for a distance of 0.1 meter, the sensor will return 1000 with a noise of 10 percent, for a distance value of 0.2 meters, the sensor will return 400 plus or minus 10 percent of noise, etc. For distance values not specified in the lookup table, the simulator will perform a linear interpolation to compute the value returned by the sensor and its associated noise. The first distance value of a lookup table must always be 0.

Note:

the ray of a sensor can be displayed in the world view by selecting **Display sensor rays** in the **File/Preferences** menu under the **Rendering** panel.

In the case of an "infra-red" sensor, the value returned by the lookup table is modified by a reflection factor depending on the color properties of the object hit by the sensor ray. This reflection factor is computed as follow: *f = 0.2 + 0.8 * red_level* where *red_level* is the level of red color of the object hit by the sensor ray. This factor is then multiplicated to the return value computed from the lookup table.

Please note that a primtive support for `DistanceSensor` nodes used for reading the red color level of a textured ground was implemented. This is useful to simulate line following behaviors. This feature is demonstrated in the `ground_color.wbt` example. In short, the ground texture should lie in a rectangular `IndexedFaceSet` node centered at (0,0,0).

The LightSensor node

The `LightSensor` node is used to model a phototransistor-like sensor which measure the level of ambiant light in a given direction. The light level measured by the `LightSensor` node is computed from each `PointLight` node in the scene, taking into account the distance between the sensor and the light, the orientation of the sensor relatively to the light, the intensity of the light (computed from its ambient intensity, intensity and color). The `LightSensor` node inherits from the `Solid` node. it includes an additional specific field:

- `lookupTable`: similar to the one of the `DistanceSensor` node except that the distance values (first column) are replaced by intensity values. This intensity value results from the sum of intensity values computed for each `PointLight` as follow:

  distance is the distance between the `LightSensor` and the `PointLight`.

  dot is the dot product between the normalized sensor direction and the normalized vector defined by the `LightSensor` location and the `PointLight` location.

  *att = attenuation.x + attenuation.y * distance + attenuation.z * distance * distance*

  *cf = color.red * color.green * color.blue*

  *intensity_value = (ambiantIntensity + intensity) * cf * dot / att*

The Camera node

The `Camera` node is used to model a robot's on-board camera. The `Camera` node inherits from the `Solid` node. The fields specific to the `Camera` node are:

- `fieldOfView`: horizontal field of view angle of the camera. The value ranges from *0* to *pi* radians. Since camera pixels are squares, the vertical field of view can be computed from the `width`, `height` and horizontal `fieldOfView`:

*vertical FOV = fieldOfView * height / width*

- `width`: width of the image in pixels.
- `height`: height of the image in pixels.
- `type`: type of the camera: ”color” or ”black and white”.

The Charger node

The `Charger` node is used to model a special kind of battery charger for the robots. A robot has to get close to a charger in order to recharge itself. A charger is not like a standard battery charger you plug to the power supply. Instead, it is a battery itself: it accumulates energy with time. It could be compared to a solar power plan loading a battery. When the robot comes to get energy, it can’t get more than the charger has currently accumulated.

The `Charger` node inherits from the `Solid` node. The fields specific to the `Charger` node are:

- `battery`: this field should contain three values: the current energy of the charger (*J*), its maximum energy (*J*) and its charging speed (*W=J/s*).
- `radius`: radius of the charging area in meters. The charging area is a disk centered on the origin of the charger coordinate system. The robot can recharge itself if its origin is in the charging area. See figure 3.6.

The Emitter node

The `Emitter` node is used to model an infra-red or radio emitter on-board a robot. You must insert the `Emitter` node into the list of children of the robot. Please note that an emitter can only emit data but it cannot receive any information. In order to enable a bi-directional communication system, a robot needs both an `Emitter` and a `Receiver` node.

The `Emitter` node inherits from the `Solid` node. The fields specific to the `Emitter` node are:

- `type`: type of the emitted signals: ”infra-red” or ”radio”.
- `range`: radius of the emission area in meters. The origin of the coordinate system of a receiver must be in this area to allow this receiver to pick up the signal.
- `channel`: channel of emission. The value is an identification number for an infra-red emitter or a frequency for a radio emitter. The receiver must use the same channel to receive the emitted signals. It can be any positive integer value.
- `baudRate`: the baudRate value is the communication speed expressed in number of bits per second.

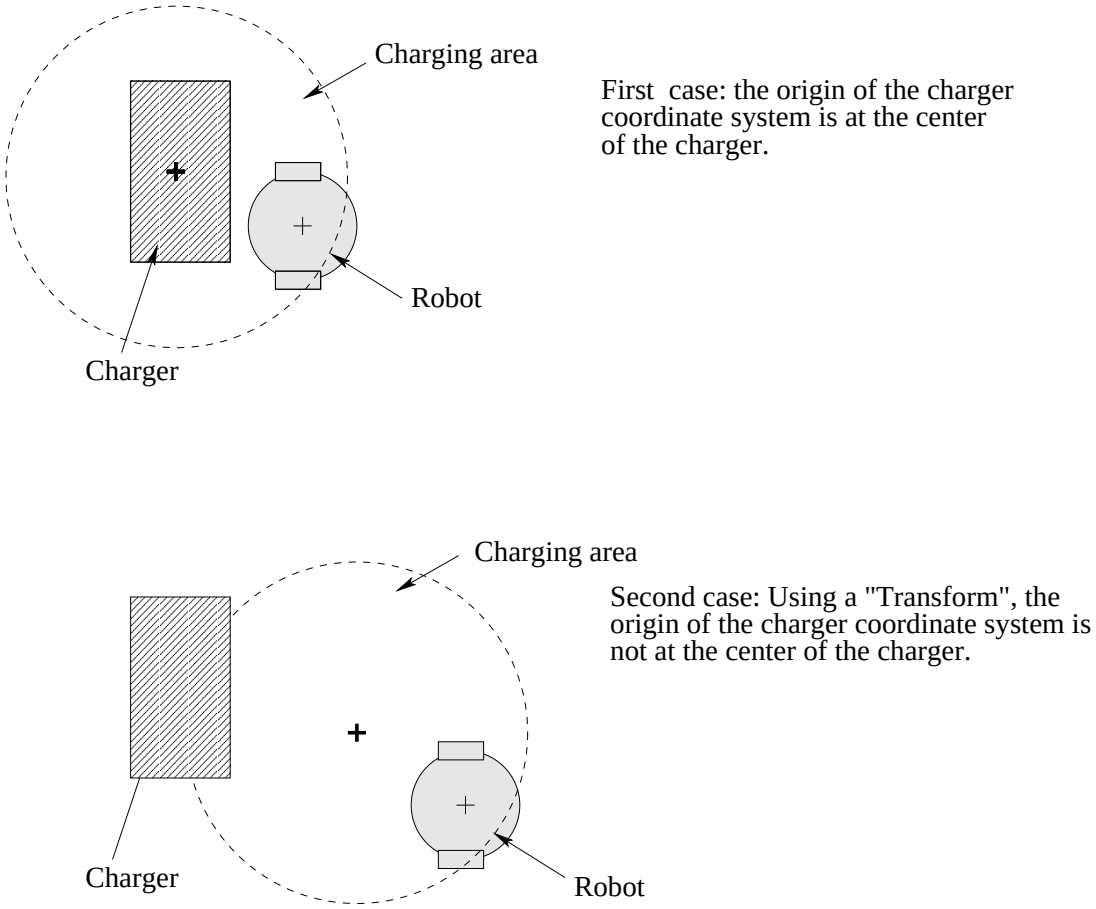


Figure 3.6: The sensitive area of a charger

- `byteSize`: the byteSize value is the number of bits used to represent one byte (usually 8, but may be more depending on whether control bits are used).
- `bufferSize`: the buffer is a memory area, its size is specified in bytes. The size of the data to be emitted cannot exceed the buffer size, otherwise data is lost. When the emitter emits the data, it flushes the buffer.

The Receiver node

The `Receiver` node is used to model an infra-red or radio receiver. A receiver, just like an emitter, is usually on-board a robot. Please note that a receiver can only receive data but it cannot emit any information. In order to enable a bi-directional communication system, a robot needs both an `Emitter` and a `Receiver` node.

The fields and values of the `Receiver` node are nearly the same as those of the `Emitter` node. As the `Emitter` node, the `Receiver` node inherits from the `Solid` node. The fields specific to the `Receiver` node are:

- `type`: type of the received signals: ”infra-red” or ”radio”.
- `channel`: channel of reception. The value is an identification number for an infra-red receiver or a frequency for a radio receiver. The emitter must use the same channel to detect the emitted signals.
- `baudRate`: the baudRate value is the communication speed expressed in bits per second. It must be the same as the speed of the emitter.
- `byteSize`: the byteSize value is the number of bits used to represent one byte (usually 8, but may be more if control bits are used). It must be the same size as the emitter buffer.
- `bufferSize`: the buffer is a memory area, its size is specified in bytes. The size of the received data can’t exceed the buffer size, otherwise data is lost. When the receiver reads the data, it flushes the buffer. If the old data has not been read when the new data is received, the former is lost.

The HyperGate node

A hypergate is defined as a cylindrical area in the world. When a robot (more precisely the origin of the robot coordinate system) enters it, it disappears and gets transferred to another world specified in the `HyperGate` node.

The `HyperGate` node inherits from the `Solid` node. The fields specific to the `HyperGate` node are:

- `url`: destination URL of the form `"wtp://host.domain.com/file#name"`.
- `radius`: radius of the transfer cylinder.
- `height`: height of the transfer cylinder.
- `maxFileSize`: maximum file size for the `Robot` node accepted by the hypergate.

  For example, an hypergate can look like an arch with the transfer cylinder lying inside the arch. See figure 3.7.

The Physics node

This node is used to specify a number of physical properties associated to a `Solid` node, like its mass, friction coefficient, energy absobtion, etc.

It was implemented to enable the modelling robot soccer systems, where a robot, or several robots can push a ball which can roll and bounce against the walls. An example of using the `Physics` node is provided in the `alice_soccer.wbt` world. The list of available fields is mentioned in the reference manual and in the `resources/nodes/Physics.wrl` file with the physical unit for each field.

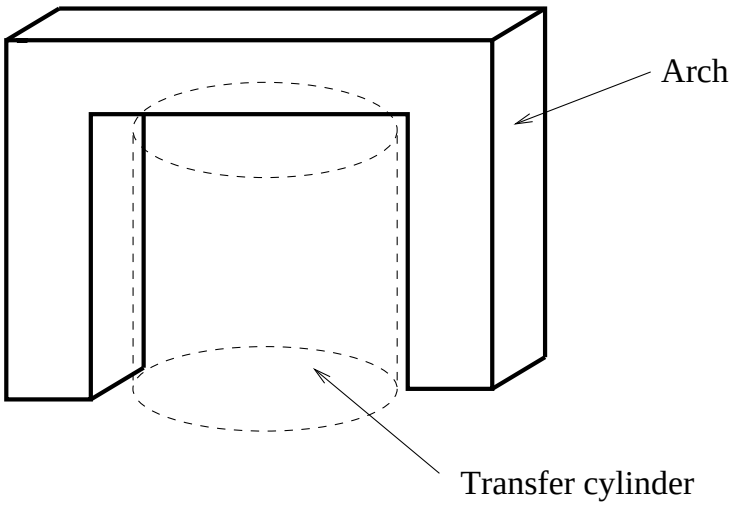


Figure 3.7: An example of an Hypergate

The Supervisor node

A supervisor is a program which controls a world and its robots. For convenience it is represented as a robot without any wheels, driven by a controller with extended capabilities which supervises the whole world. A world cannot have more than one supervisor.

The `Supervisor` node inherits from the `Solid` node. Its other fields include some of the `DifferentialWheels` node fields:

- `controller`
- `synchronisation`
- `battery`: usually meaningless for a `Supervisor` node.
- `cpuConsumption`: usually meaningless for a `Supervisor` node.

The TouchSensor node

The `TouchSensor` node is used to model bumper sensors. A bumper sensor will detect the collision with any `Solid` object in the world, including other `DifferentialWheels` nodes. Collision detection is based upon the `boundingObject` field of the `TouchSensor` node and the `boundingObject` field of other `Solid` nodes. The `TouchSensor` node inherits from the `Solid` node. It includes two additional specific fields:

- `lookupTable`: similar to the one of the `DistanceSensor` node.
- `type`: type of sensor: ”bumper”.

Note:

only the ”bumper” type is currently supported, but other types, including ”button”, ”force” or ”whisker” are likely to be implemented in a forthcoming version of Webots.

3.3.4 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 VRML 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 VRML file.

The Webots header is:

```
#VRML_SIM V3.0 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).

Chapter 4

Tutorial: Modelling 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 will show you how to model a camera on this simple robot. The third example will show you how to build a virtual Pioneer 2™ robot from ActivMedia Robotics. The fourth part will explain how to work with robots from K-Team.

4.1 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 4.1), controlled by a program inspired by a Braitenberg algorithm, in a simple environment surrounded by a wall.

4.1.1 Environment

We just want to have a simple world with a surrounding wall. We will represent this wall using an `Extrusion` node in the tree editor. The coordinates of the wall are shown in figure 4.2.

First, go to the **File** menu, **New** item to open a new world. Then open the tree editor (in the **Edit** menu). We are going to change 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 0.5 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`.

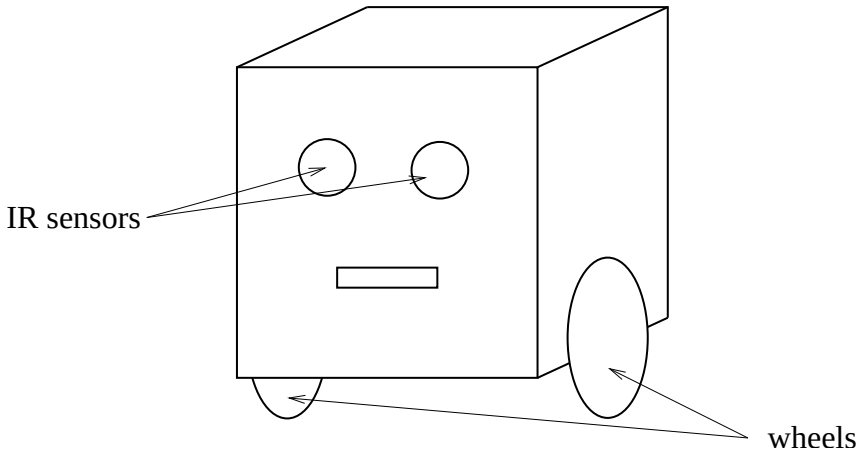
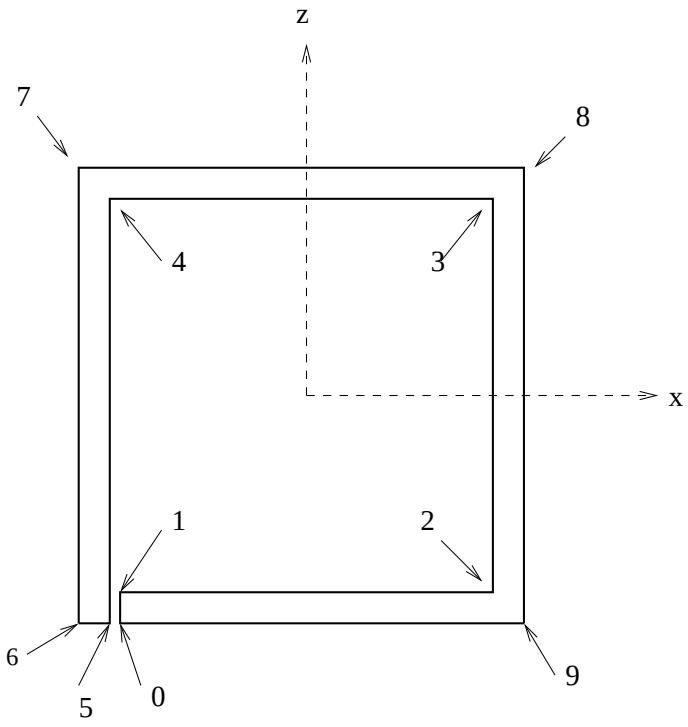


Figure 4.1: The *kiki* robot

2. Select the `PointLight` node, copy and paste it. In this new `PointLight` node, type -0.5 0.5 0.5 in the `location` field.
3. Repeat this copy/paste 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.

Secondly, let us create the wall:

1. Select the last `Solid` node and click on the **insert after** button.
2. Choose a `Solid` node.
3. Fill in the text fields with your desired text, e.g., ”wall” for the name.
4. Select the `children` field and **Insert after** a `Shape` node.
5. **Insert** a new `Appearance` node in the `appearance` field. **Insert** a new `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.
6. Now insert an `Extrusion` node in the `geometry` field of the `Shape`.
7. Set the wall corner coordinates in the `crossSection` field and set `convex` to `FALSE`.
8. In the `spine` field, write that the wall ranges between 0 and 0.1 along the y axis.
9. As we want our robot to detect the wall, we have to fill in the `boundingObject` field. The bounding object can have exactly the same shape as the wall, so **insert** a DEF at the level of `geometry Extrusion`: WALL. Then, in the `boundingObject` field, **insert** `USE WALL`.



(x,z) coordinates:

0 (−0.489, −0.5)
1 (−0.489, −0.49)
2 (0.49, −0.49)
3 (0.49, 0.49)
4 (−0.49, 0.49)
5 (−0.49, −0.5)
6 (−0.5, −0.5)
7 (−0.5, 0.5)
8 (0.5, 0.5)
9 (0.5, −0.5)

Figure 4.2: The *kiki* world

10. Close the tree editor, save your file and look at the result.

The wall in the tree editor is represented in figure 4.3, while the same wall in the world editor is visible in figure 4.4

4.1.2 Robot

The aim of this subsection is to model the *kiki* robot. This robot is made up of a `DifferentialWheels` node, in which we find 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 4.5.

To model the body of the robot:

1. Open the tree editor.
2. Select the last `Solid` node.
3. **Insert after** a `DifferentialWheels` node, give it a name: ”kiki”.

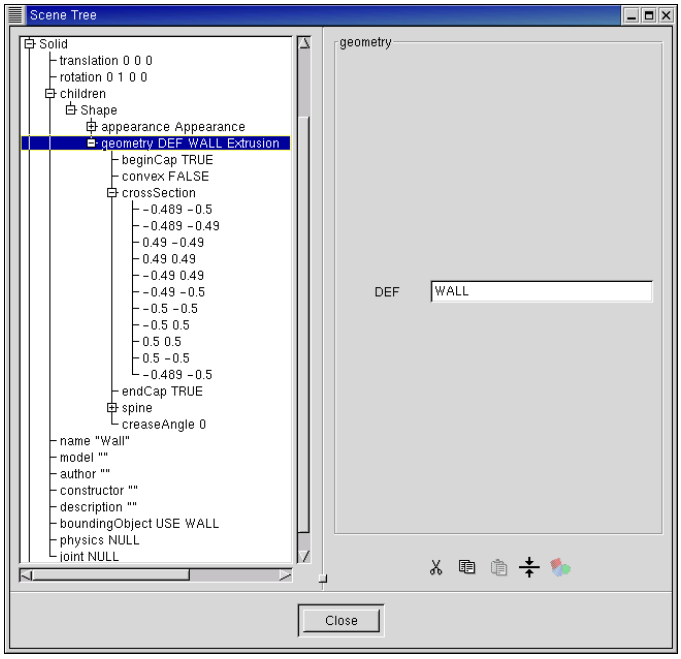


Figure 4.3: The wall in the tree editor

4. In the `children` field, first introduce a `Transform` node that will contain a box shape. 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 dimension 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 4.6)

To model the left wheel of the robot:

1. Select the previous `Transform` 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*. Moreover, a wheel is 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 a `Shape` with a `Cylinder` in its `geometry` field. The dimensions of the cylinder are 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 false, 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].

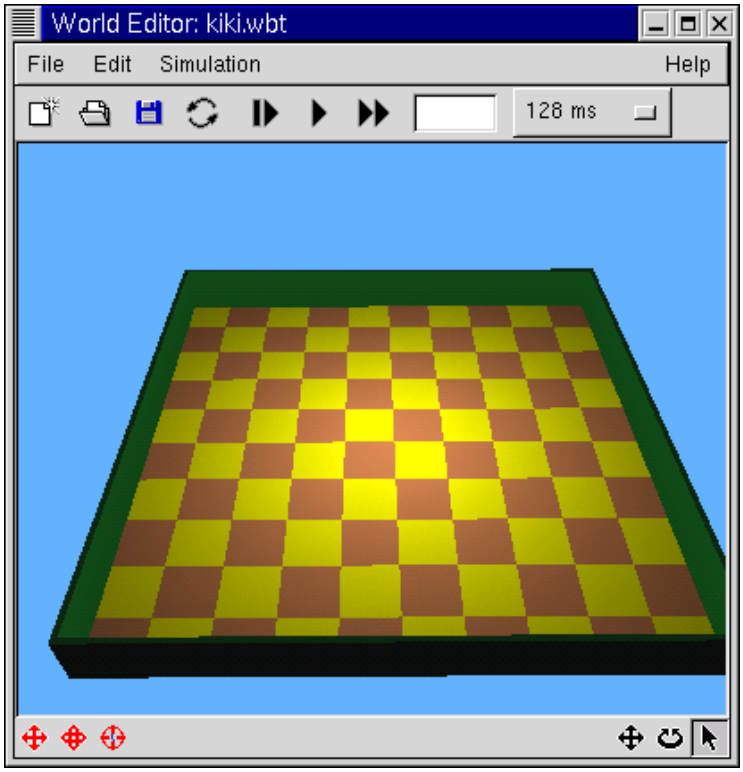


Figure 4.4: The wall in the world window

5. Give a `DEF` name to your `Transform`: WHEEL; notice that you positionned 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 4.7 and figure 4.8.

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, towards the front of the robot.

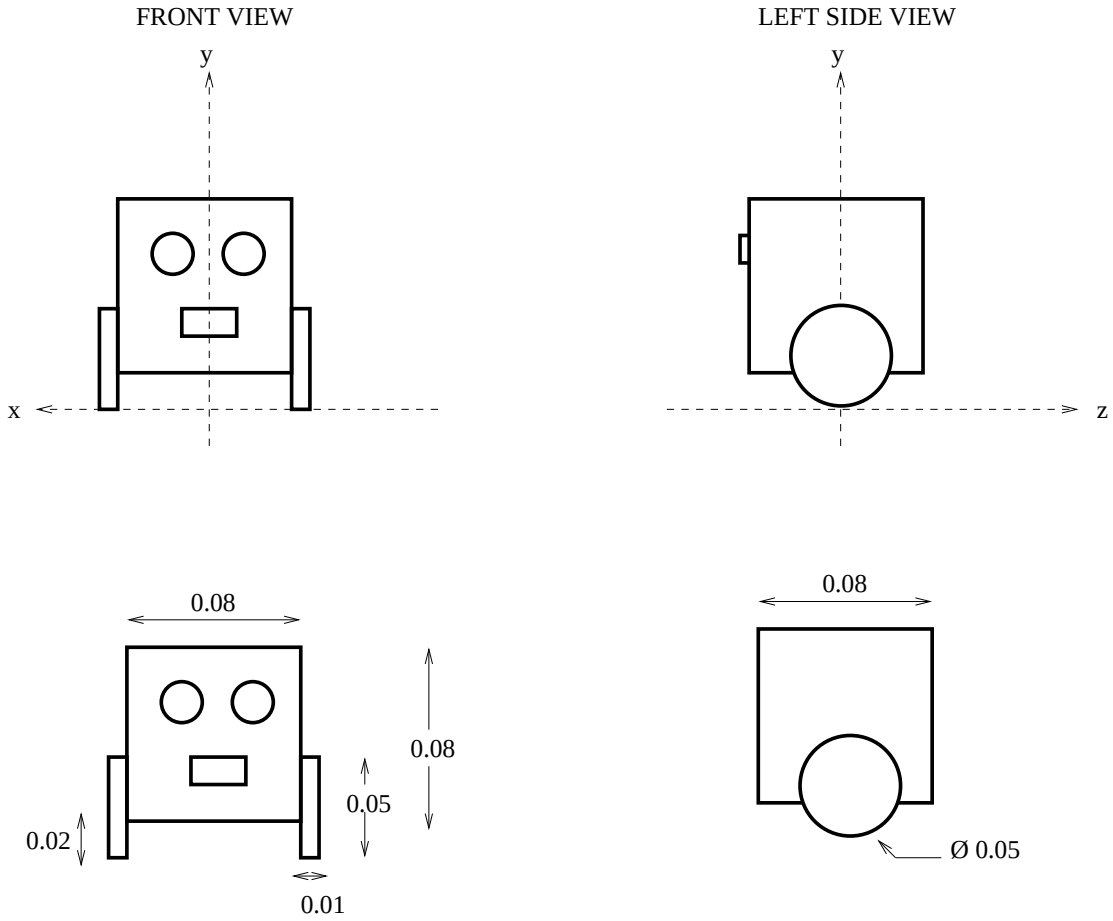


Figure 4.5: Coordinate system and dimensions of the *kiki* robot

1. In the `chidren` of the `DifferentialWheels` node, **insert after** a `DistanceSensor` node.
2. Type the name "ir0". It will be used by the controller program.
3. Now, we will attach a cylinder shape to this sensor. In the `chidren` of the `DistanceSensor` node, **insert after** a `Tranform` node. Give a `DEF` name to it: INFRARED, which you will use for the second IR sensor.
4. In the `chidren` of the `Tranform` node, **insert after** a `Shape` node. Choose an appearance and **insert** a `Cylinder` in the `geometry` field. Type 0.004 for the height and 0.08 for the radius.
5. Set the rotation for the `Tranform` 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.

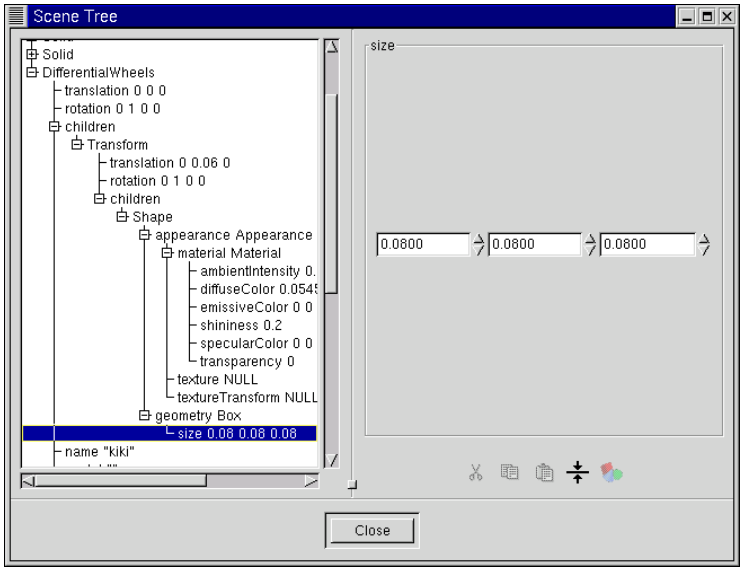


Figure 4.6: Body of the *kiki* robot: a box

In order to have the ray directed towards 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 4.9. These values are:

```
lookupTable [ 0     1024  0,
              0.05  1024  0,
              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 4.10 and figure 4.11.

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 *(2n̂) * (2n̂)* 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

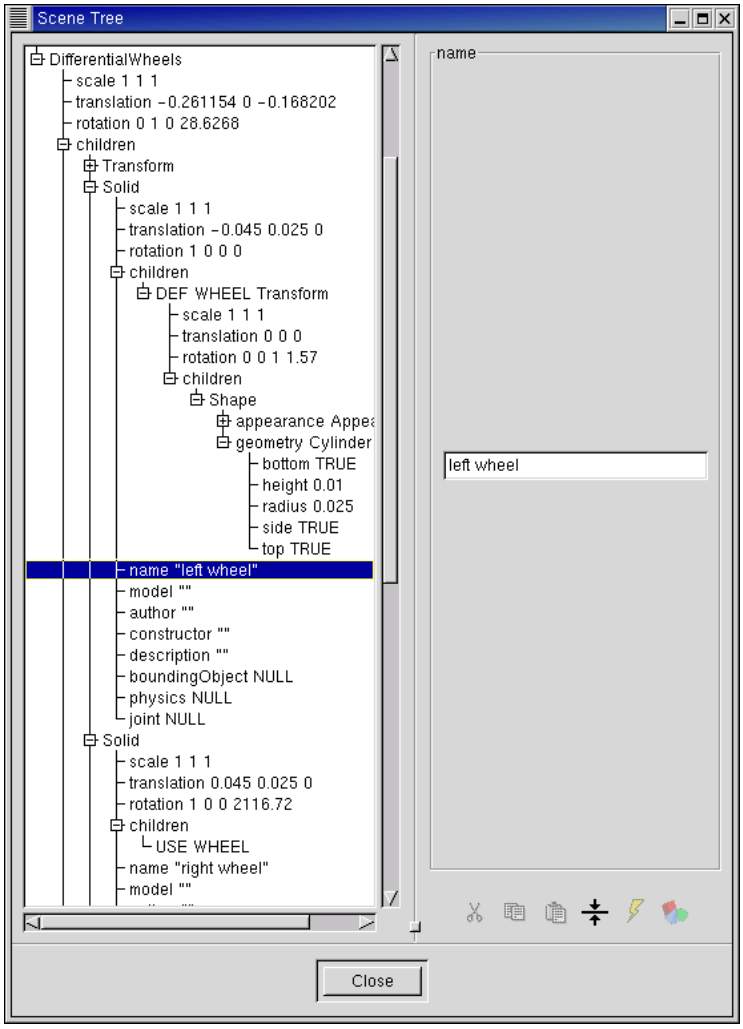


Figure 4.7: Wheels of the *kiki* robot

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:

1. Select the last `DistanceSensor` node and **insert after** a `Shape` node.
2. In the `appearance` field, **insert** an `Appearance` node. In the `texture` field of this node, **insert** an `ImageTexture` node with the following URL: `"kiki/kiki.png"`.
3. In the `geometry` field, **insert** 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,
```

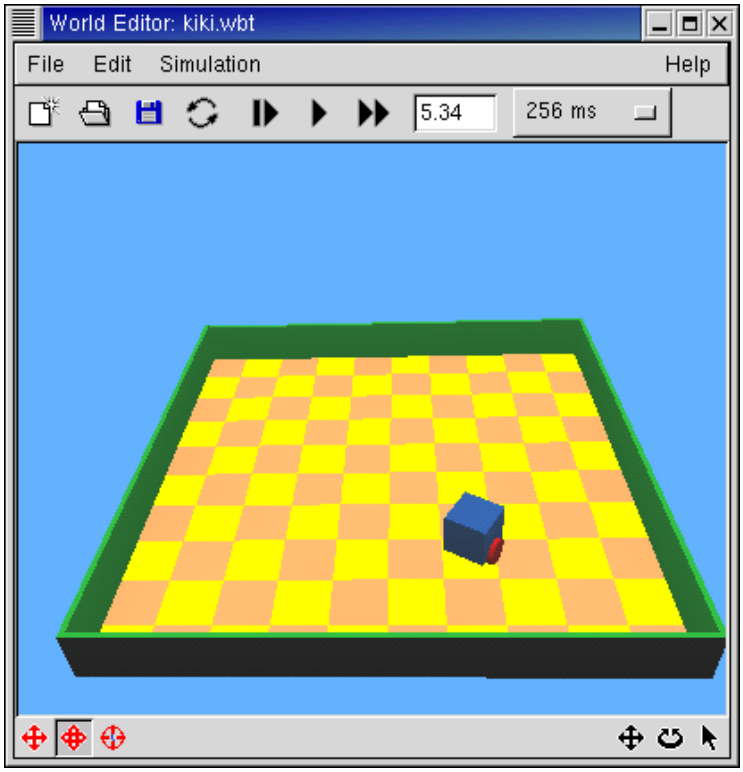


Figure 4.8: Body and wheels of the *kiki* robot

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

and **insert after** the `coordIndex` field the values 0, 1, 2, 3, -1.

4. In the `texCoord` field, **insert** a `TexureCoordinate` 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.

5. The texture values are shown in figure 4.12.

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

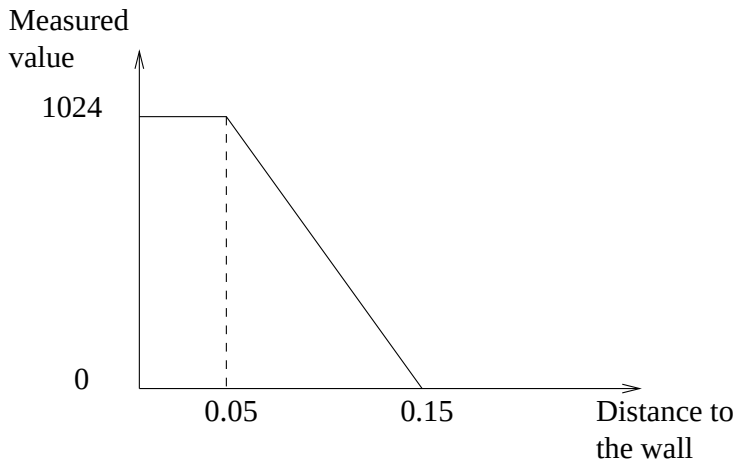


Figure 4.9: Distance measurements of the *kiki* sensors.

1. In the `controller` field, type the name ”simple”. It will be used by the controller program.
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. **Insert** 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 4.5).
4. In the `wheelRadius` field, enter the radius of the wheels: 0.025.
5. Values for other fields are shown in figure 4.13 and the finished robot in its world is shown in figure 4.14.

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

4.1.3 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 <robot.h>
#include <differential_wheels.h>
#include <distance_sensor.h>
```

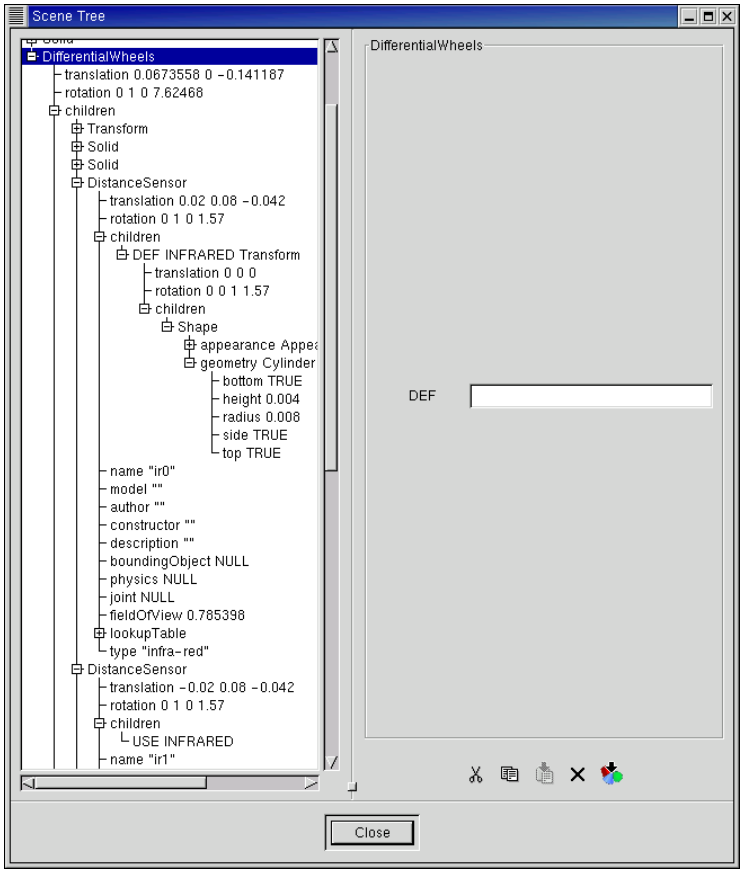


Figure 4.10: The DistanceSensor nodes of the *kiki* robot

```
#define SPEED 100

static DeviceTag ir0,ir1;

void reset(void) {
  ir0 = robot_get_device("ir0");
  ir1 = robot_get_device("ir1");
  // g_print("ir0=134530019 ir1=1076052308\n",ir0,ir1);
}

int main() {
  gint16 left_speed,right_speed;
  guint16 ir0_value,ir1_value;

  robot_live(reset);
  distance_sensor_enable(ir0,64);
  distance_sensor_enable(ir1,64);
  for(;;) {        /* The robot never dies! */
```

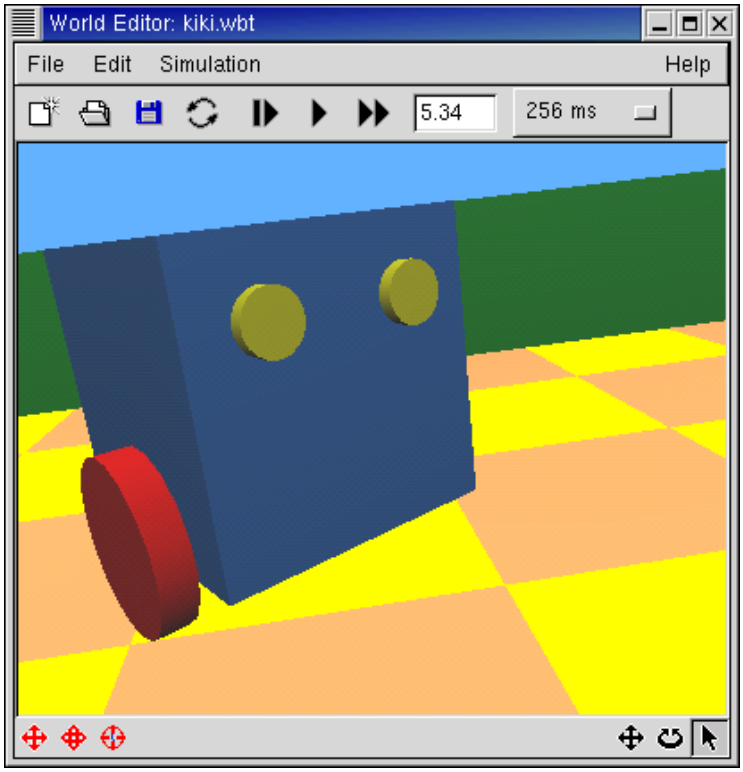


Figure 4.11: The *kiki* robot and its sensors

```
  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;
  }
  /* Set the motor speeds */
  differential_wheels_set_speed(left_speed,right_speed);
  robot_step(64); /* run one step */
}
return 0;
```

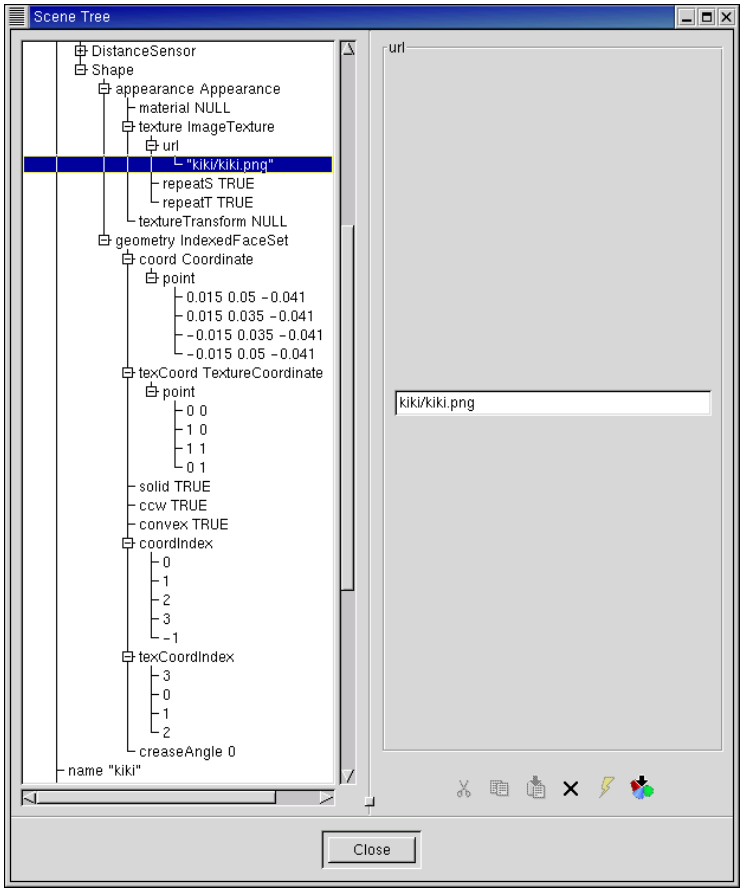


Figure 4.12: Defining the texture of the *kiki* robot

4.2 My second world: a *kiki* robot with a camera

The camera to be modelled 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 4.15.

Try modelling 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.

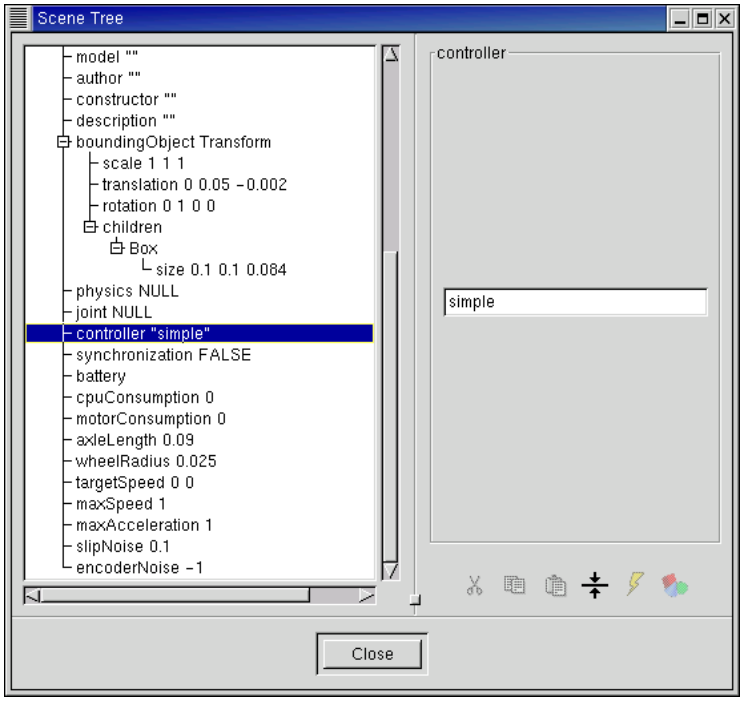


Figure 4.13: The other fields of the DifferentialWheels node

4.3 My third world: 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 `controllerss/pioneer2` files for the world and controller details.

4.3.1 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 4.16.

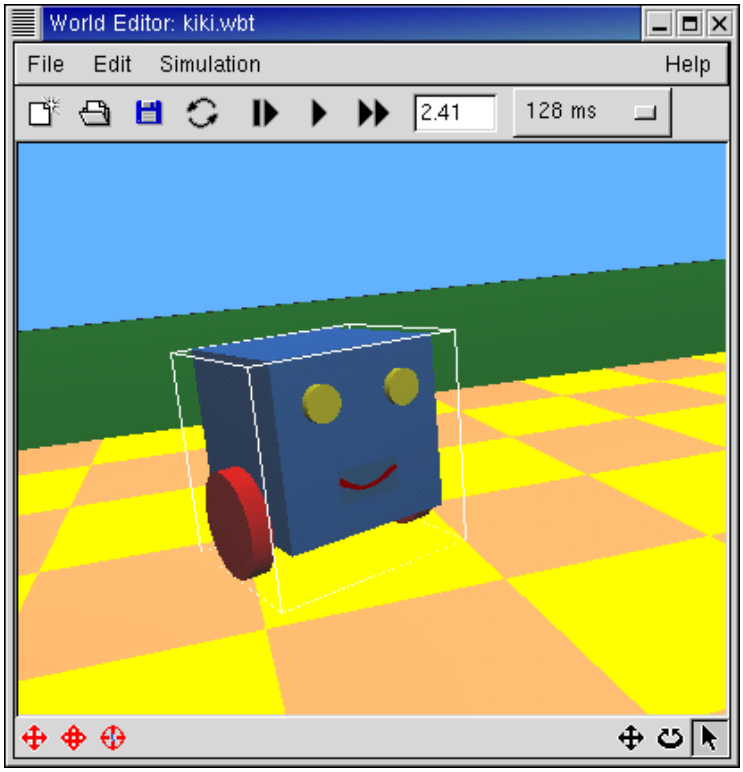


Figure 4.14: The *kiki* robot in its world

4.3.2 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 DX™ robot is depicted in figure 4.17.

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 4.18 for the coordinates of the `Extrusion`.

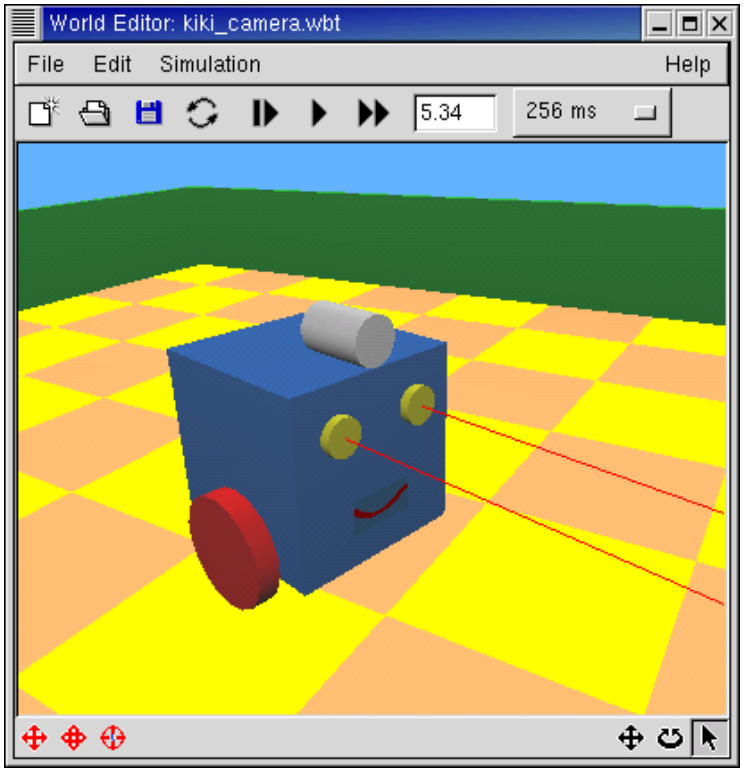


Figure 4.15: The *kiki* robot with a camera

2. for the top plate: a `Shape` node with a `geometry Extrusion`. See figure 4.19 for the coordinates of the `Extrusion`.

3. 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 4.20 for the wheels dimensions.

4. for the rear wheel: a `Transform` node containing a `Shape` node with a `geometry Cylinder` , as shown in figure 4.21

5. for the sonar supports: two `Shape` nodes with a `geometry Extrusion`. See figure 4.22 for the `Extrusion` coordinates.

6. 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 4.23 and the text below for more explanation.

Modelling 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

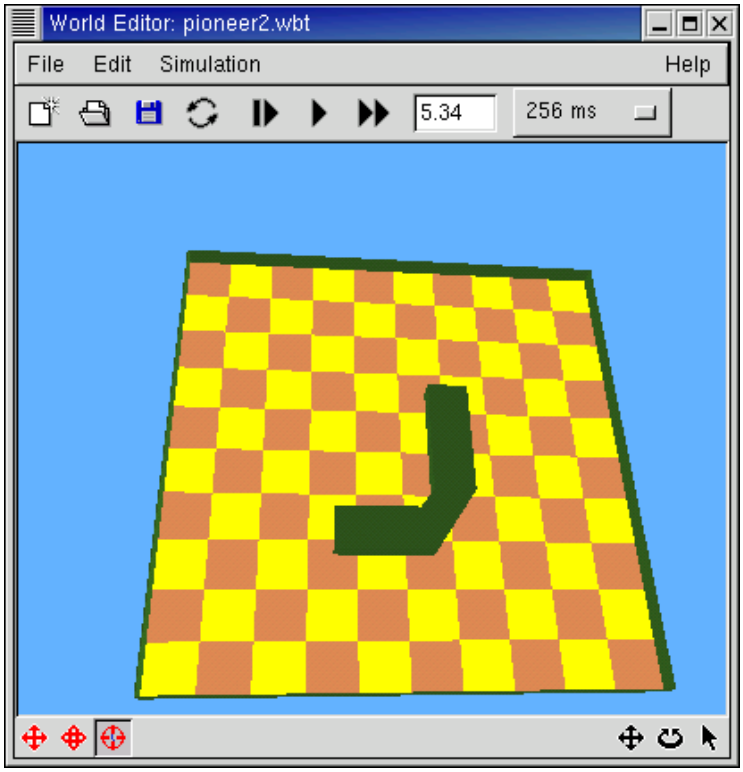


Figure 4.16: The walls of the Pioneer 2^{TM} robot world

robot (see figure 4.23). The angles between the sonars and the initial position of the `DEF SONAR Transform` are shown in figure 4.24. 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 modelled 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 by the controller.

To finish modelling the Pioneer 2^{TM} robot, fill in the remaining fields of the `DifferentialWheels` node as shown in figure 4.25.

4.3.3 Controller

The controller of the Pioneer 2^{TM} 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 2^{TM} is not circular some tricks are used, such as making the robot go backwards 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`.

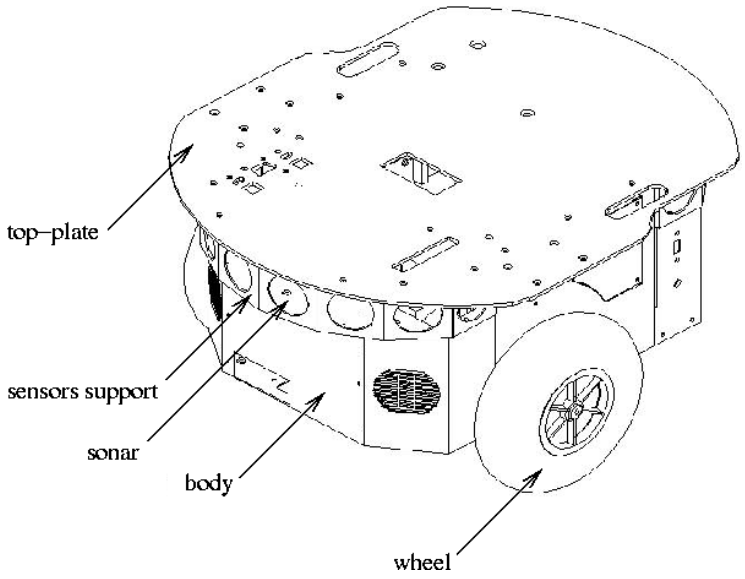
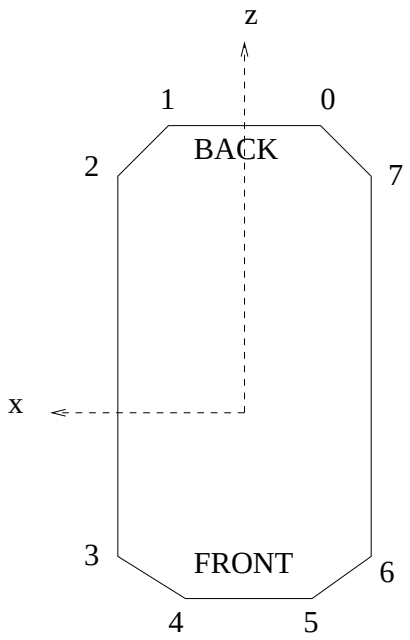


Figure 4.17: The Pioneer 2 DX™ robot



Coordinates of the crossSection field of the extrusion node:
0: x=−0.1, z=0.215
1: x=0.1, z=0.215
2: x=0.135, z=0.185
3: x=0.135, z=−0.095
4: x=0.08, z=−0.11
5: x=−0.08, z=−0.11
6: x=−0.135, z=−0.095
7: x=−0.135, z=0.185

$0.059 < y < 0.234$

Figure 4.18: Body of the Pioneer 2™ robot

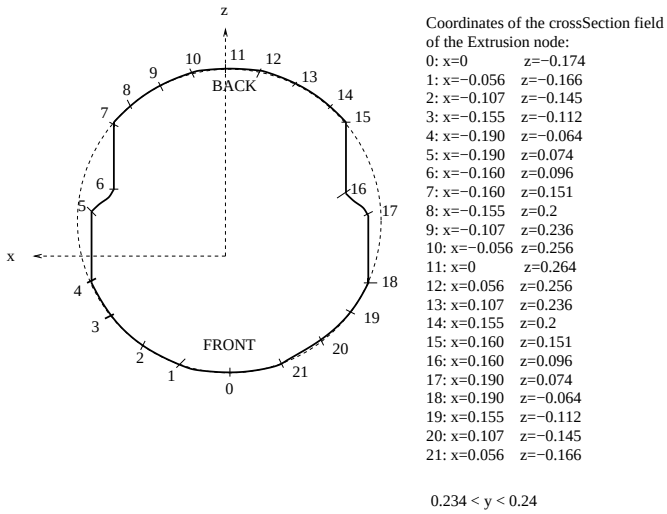


Figure 4.19: Top plate of the Pioneer 2^{TM} robot

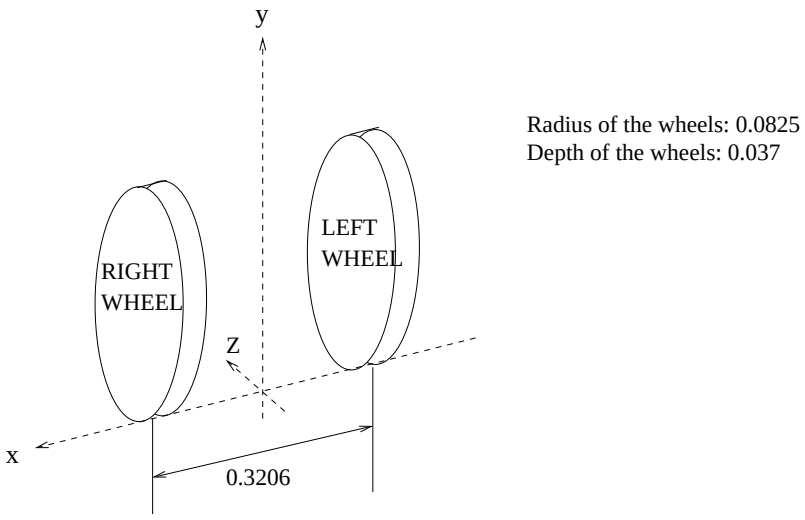


Figure 4.20: Wheels of the Pioneer 2^{TM} robot

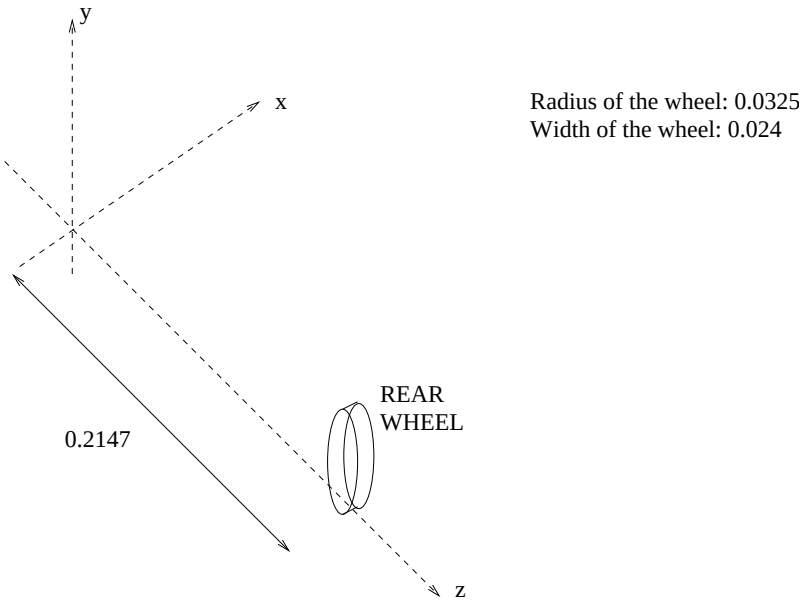
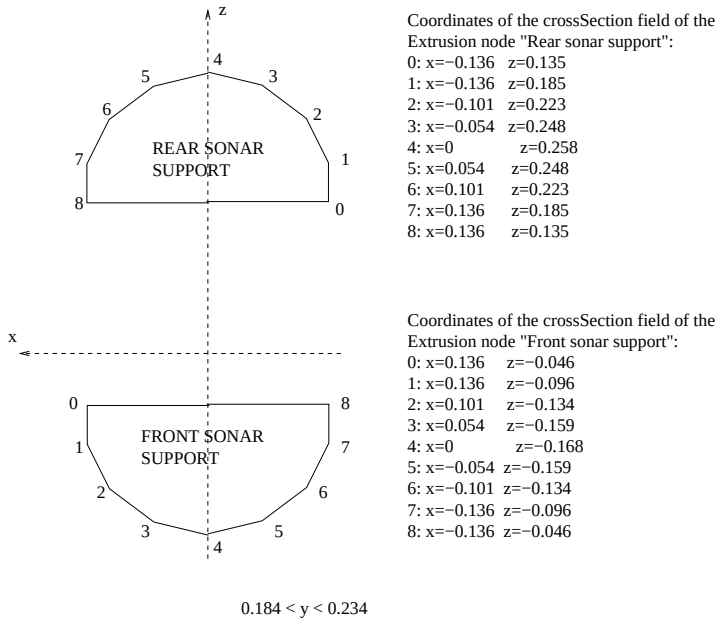


Figure 4.21: Rear wheel of the Pioneer 2^{TM} robot



Coordinates of the crossSection field of the Extrusion node "Rear sonar support":
0: x=−0.136 z=0.135
1: x=−0.136 z=0.185
2: x=−0.101 z=0.223
3: x=−0.054 z=0.248
4: x=0 z=0.258
5: x=0.054 z=0.248
6: x=0.101 z=0.223
7: x=0.136 z=0.185
8: x=0.136 z=0.135

Coordinates of the crossSection field of the Extrusion node "Front sonar support":
0: x=0.136 z=−0.046
1: x=0.136 z=−0.096
2: x=0.101 z=−0.134
3: x=0.054 z=−0.159
4: x=0 z=−0.168
5: x=−0.054 z=−0.159
6: x=−0.101 z=−0.134
7: x=−0.136 z=−0.096
8: x=−0.136 z=−0.046

Figure 4.22: Sonar supports of the Pioneer 2^{TM} robot

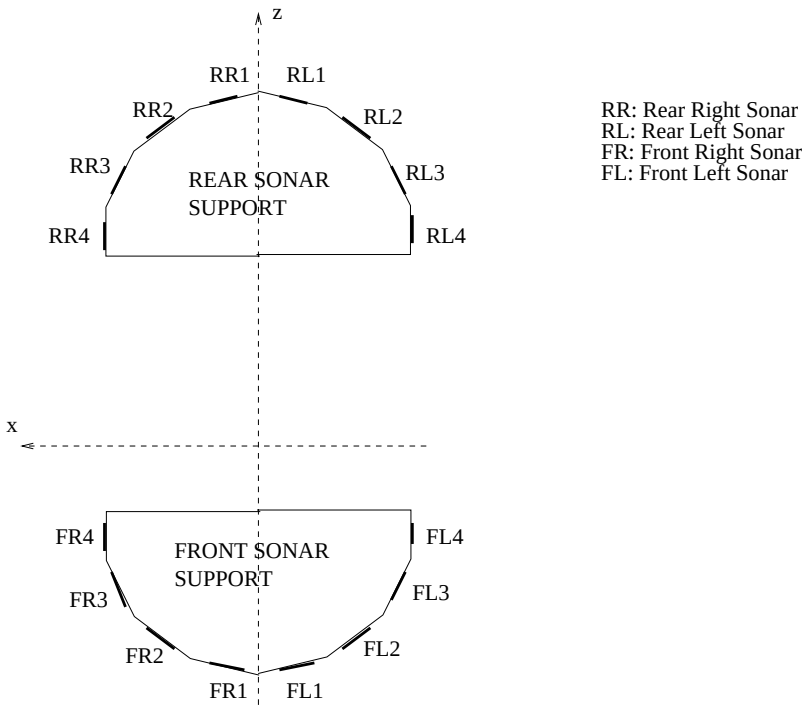


Figure 4.23: Sonars location on the Pioneer 2^{TM} robot

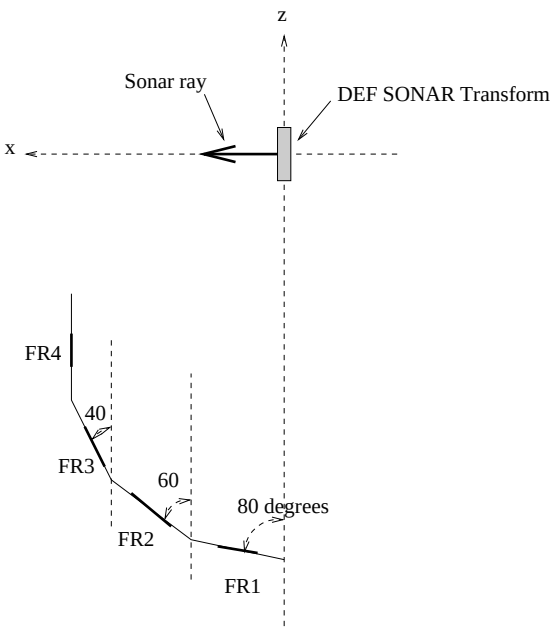


Figure 4.24: Angles between the Pioneer 2^{TM} sonar sensors

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 -0.11	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 4.1: Translation and rotation of the Pioneer 2™ DEF SONAR Transforms

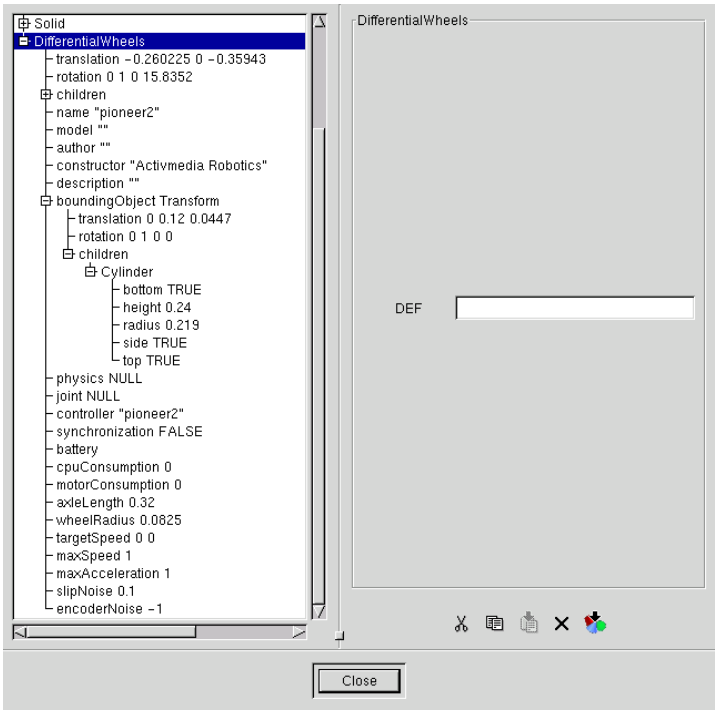


Figure 4.25: Some fields od the Pioneer 2™ `DifferentialWheels` node

Chapter 5

Robot and Supervisor Controllers

5.1 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.

5.2 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 have several alternatives to the Makefile: They can use a Dev-C++ project or a Microsoft Visual C++ project, as examplified in the Webots `controllers/braiten` directory.

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

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

5.3 Webots Execution Scheme

5.3.1 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 identify the devices of the robot (see section 5.4). Then an endless loop (usually implemented as a `for(;;) { }` statement) runs the controller continuously until the simulator decides to terminate it. This endless loop must contain at least one call to the `robot_step` function which asks the simulator to advance the simulation time a given number of milliseconds, thus advancing the simulation. Before calling `robot_step`, the controller can enable sensor reading and set actuator commands. Sensor data can be read immediately after calling `robot_step`. Then you can perform your calculations to determine the appropriate actuator commands for the next step.

5.3.2 From the point of view of Webots

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.

5.3.3 Synchronous versus Asynchronous controllers

Each robot (`DifferentialWheels` or `Supervisor`) may be either 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).

5.4 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.

5.5 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 (see `differential_wheels` for an example). It is usually called in the endless loop with different computed values at each step.
3. *run*: this is done by the `robot_step` function inside the endless loop.

5.6 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!

Chapter 6

Tutorial: Using the Khepera^{TM} robot

The goal of this chapter is to explain you 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 protocole 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.

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** menuitem 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.

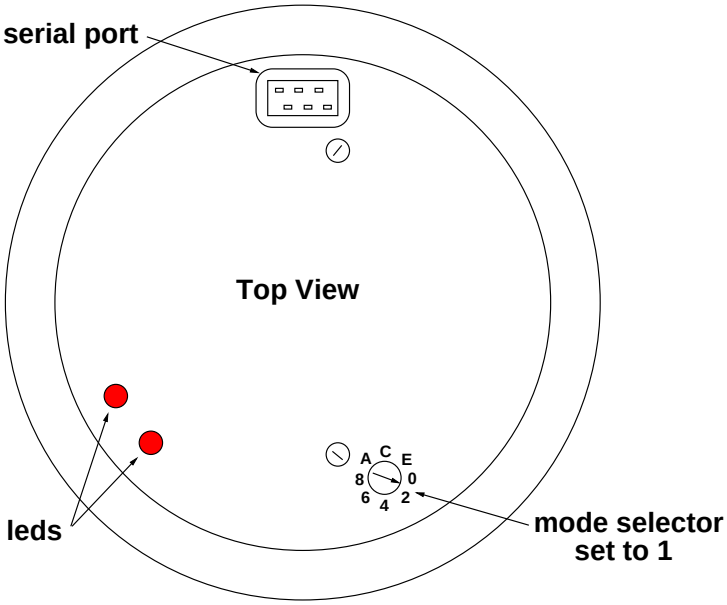


Figure 6.1: Khepera II mode selection

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 **Simulation Robot View** menuitem. 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).

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

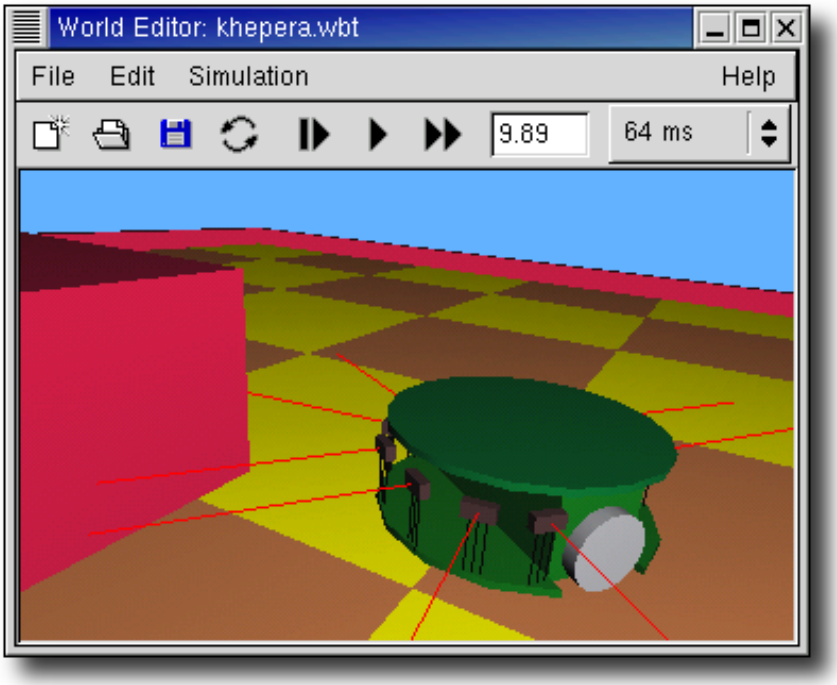


Figure 6.2: Khepera example world

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 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.

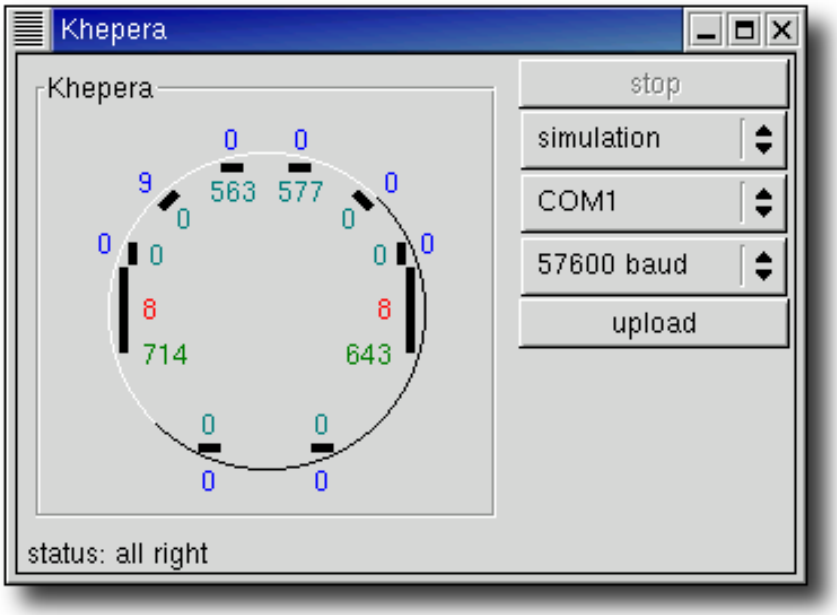


Figure 6.3: Khepera Controls

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.

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.

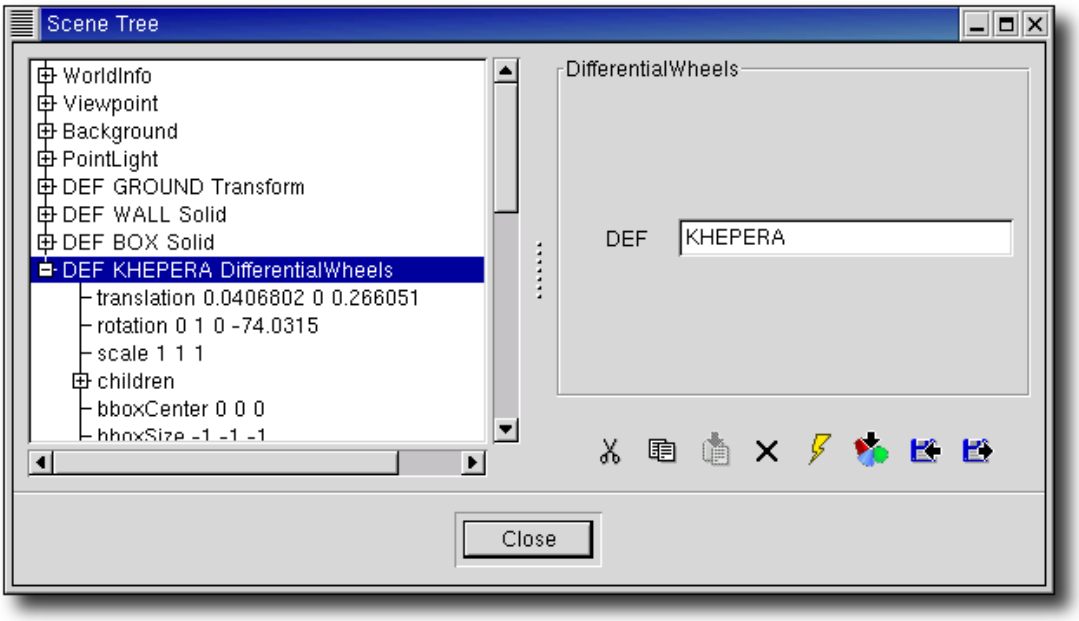


Figure 6.4: Scene tree window for the Khepera world

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 notived 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 <transfer/khepera.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() {
  gint16 left_speed=0,right_speed=0;
  guint16 ds1_value,ds2_value,ds3_value,ds4_value,ls2_value,ls3_value;
  guint32 left_encoder,right_encoder;

  khepera_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 backwards */
    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, then 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 necesseray 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 Transfering 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 **stop** / **go** button to prevent or allow you robot to move. 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 protocole 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 a executable file for the Khepera micro-controller from your C source file. In order to produce such an executable, you can use the `Makefile.kros` provided within the `khepera` controller directory. From Linux, just type:

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

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
```

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. 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.

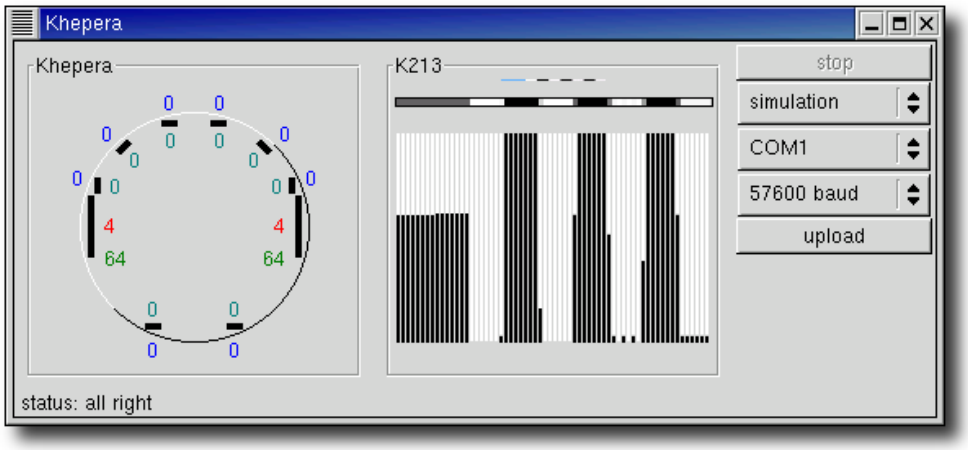


Figure 6.5: Khepera K213 controls

6.6.2 The Gripper turret

figure 6.6 shows the `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 modelling 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.

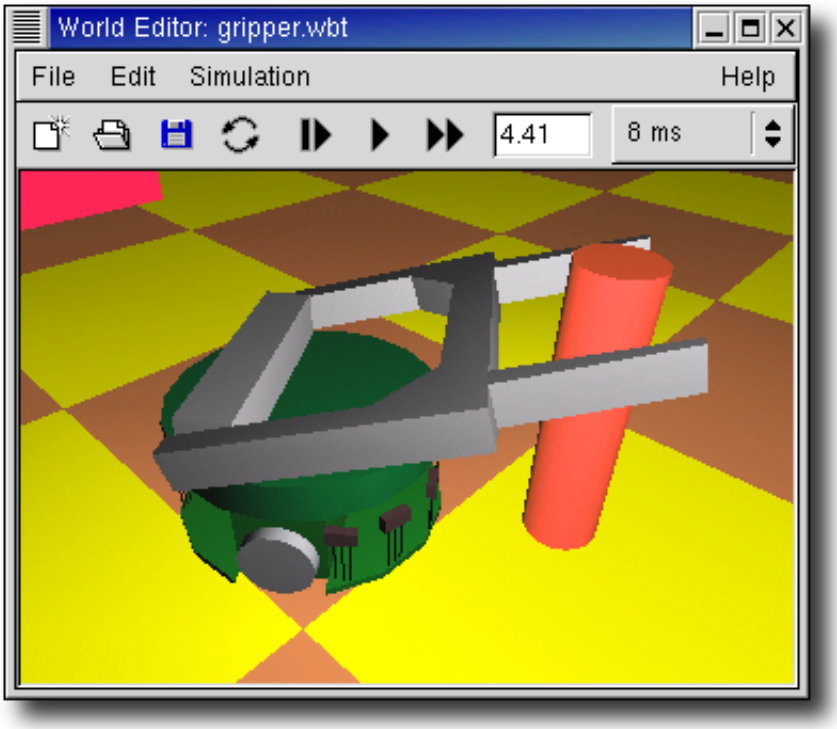


Figure 6.6: Khepera Gripper

However, the current version of Webots do not yet support the transfer of controllers using a gripper to the real Khepera robot. But if this feature is very important for you, please contact Cyberbotics Ltd. and we will see how we can help you implement this feature. Webots is an open platform and Cyberbotics people are open minded, so there is always a solution to any problem. Cyberbotics's support can be reached by e-mail a the following address:

<support@cyberbotics.com>

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 6.7. The example can be found in `worlds/koala.wbt`.

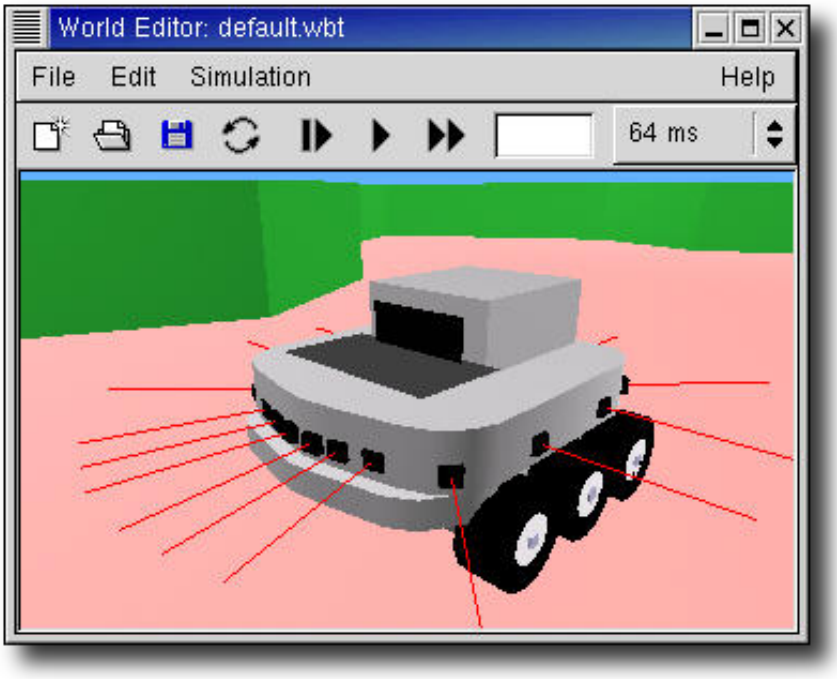


Figure 6.7: The Koala robot

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 6.8). The example can be found in `worlds/alice.wbt`.

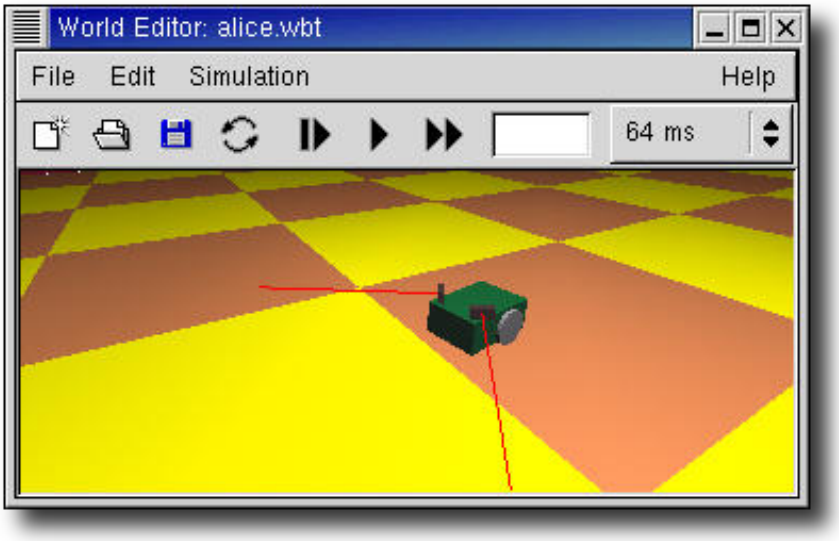


Figure 6.8: The Alice robot

Chapter 7

ALife Contest

A programming contest based on Webots in organized permanently on the Internet. The web site of the contest[1] may provide more up to date information about it. ALife stands for "Artificial Life".

7.1 Previous Editions

This is 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.

7.2 Rules

7.2.1 Subject

Two robots are roaming a maze-like environment (see figure 7.1), looking for energy. Energy is provided by chargers (see figure 7.2). However, chargers are scattered all around the environment and it is not so easy for robots to find them. Moreover, once used by a robot, a charger will be unavailable for a while (see figure 7.3). 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.

[1] `http://www.cyberbotics.com/contest/`

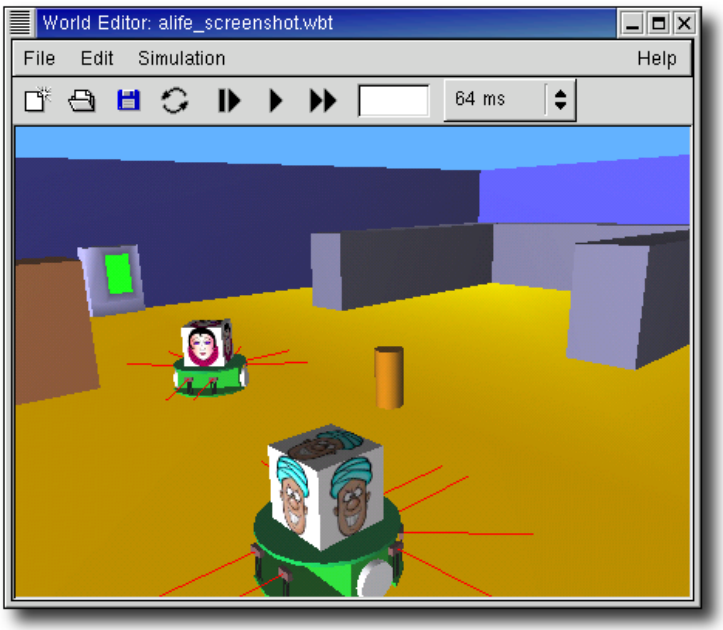


Figure 7.1: The world used in the contest

7.2.2 Robot Capabilities

All robots have the same capabilities. They are based on a model of Khepera 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, analysing the camera image is a crucial issue in developing an efficient robot controller.

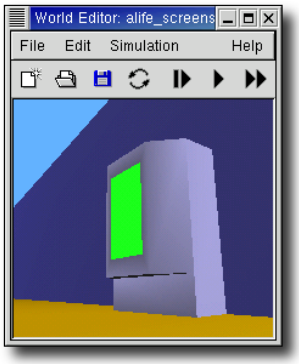


Figure 7.2: A charger full of energy

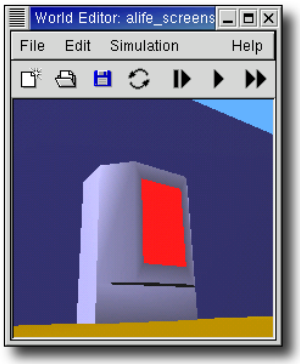


Figure 7.3: An empty charger

7.2.3 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 executables files (`.class` files) can be easily shared amongst competitors without disclosing source code. Beware, that very good Java decompilers exists and that it may be possible (although difficult and tricky) for a cheating competitors to restore your code from your `.class`. If we encounter such problems, we will disable the Java binary code sharing.

They 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.

7.2.4 Scoring Rule

Once submitted on the web site, a robot is assigned a score of 10 points. Then, it will perform continuously a number of matches against other robots until its score reaches 0. At that point the robot will be eliminated from the contest. Scores are floating point values. When a robot wins a match it draws points from its opponent. The amount of points transfered from the loser to the winner is computed as follow:

1 + abs(winner_score - loser_score)/winner_score

If the amount of points to be transfered is higher than the score of the loser, then the loser is eliminated from the competition and the winner only receives the score of the looser, no more.

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 score remains unchanged. The next matches are run using the new version.

7.2.5 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.

7.2.6 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 7.4). Prizes for the second and third position are not yet determined.

Figure 7.4: First prize: a Khepera II robot and a Webots PRO package.

7.3 Web Site

The web site of the contest[2] allows you to view matches runing 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

[2] `http://www.cyberbotics.com/contest/`

possible to visit the home page of each robot engaged in the contest, including a small description of the robot's algorihtm, the flag of the robot and possibly the e-mail of the author. You can even download the Java binary controller (`.class` file) of each robot. This can be useful to understand why a robot performs so well.

7.4 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.

7.4.1 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 page[3]. 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 site[4] for free. Please use the version 1.4 of the SDK. Follow the instructions from Sun to install the SDK.

7.4.2 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 a the source code of the programs.

On Windows, you may need to edit the `ALife0.bat` and `ALife1.bat` files to set correct paths to the Webots directory and possibly to the `java` executable.

7.4.3 Creating your own robot controller

The simpliest way to create your own robot controller is to start from the existing `ALife0` or `ALife1` controllers.

[3] `http://www.cyberbotics.com/products/webots/download.html`
[4] `http://java.sun.com/j2se/1.4/download.html`

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 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.
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 loosing 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 modifiy the source code of the controllers in your `controllers` directory, recompile them and test them with Webots.

Modifiying 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 recommanded 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.

7.4.4 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 page[5], 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)
- 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.

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.

[5] `http.//www.cyberbotics.com/contest`

7.5 Developers' Tips and Tricks

This section contains some hints to develop efficiently an intelligent robot controller.

7.5.1 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.

7.5.2 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 window, opening any file for writing or reading and doing any networking stuff.

7.5.3 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 towards 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 rapidely 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 faciliting 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...