Webots 3.0
User Guide

`www.cyberbotics.com`

June 19, 2001

Trademark information

Foreword

Webots is a three-dimensional mobile robot simulator. It was originaly 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 or Java 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 (however the transfer to these real robots is no longer possible).

The GUI of Webots version 2 has been replaced by GTK+, an Open Source Free Software GUI toolkit.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 3.0.

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

List of Figures

List of Tables

Chapter 1

Installing Webots

1.1 Hardware requirements

Webots is available for RedHat 7.0 Linux i386, Windows 95, Windows 98, Windows 2000, Windows NT and Windows ME. Other versions of Webots for other UNIX / X11 systems (Solaris, Linux PPC, Irix) are available upon request.

OpenGL hardware acceleration is supported on Windows and in some Linux configurations. It may also be available on other UNIX / X11 systems.

1.2 Registration procedure

1.2.1 Webots License

Starting with Webots 3.0, 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 provide copies of it to any third party in any way, including publication of 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

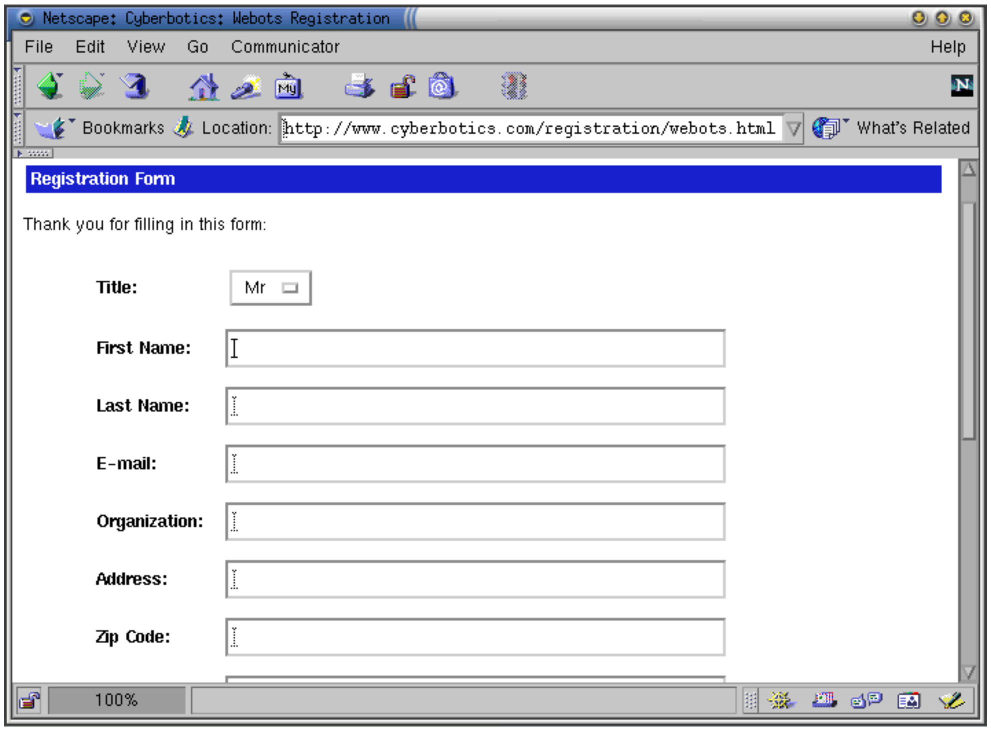
`license@cyberbotics.com`.

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 on the website of Cyberbotics. You will then receive an e-mail containing the `webots.key` file corresponding to your license. The form can be found at the following web address (see figure 1.1):

`http://www.cyberbotics.com/registration/webots.html`


Netscape: Cyberbotics: Webots Registration
File Edit View Go Communicator Help
Bookmarks Location: http://www.cyberbotics.com/registration/webots.html What's Related
100%

Registration Form

Thank you for filling in this form:

Title:	Mr
First Name:	
Last Name:	
E-mail:	
Organization:	
Address:	
Zip Code:	

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:`.

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 register 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 7.0 Linux i386

1. Log on as `root`.
2. Insert the Webots CD-ROM and mount it (this might be automatic).
```
mount /mnt/cdrom
rpm -Uvh /mnt/cdrom/libraries/miniGLU-3.0-1.i386.rpm
rpm -Uvh /mnt/cdrom/libraries/solid-2.0.1-1.i386.rpm
rpm -Uvh /mnt/cdrom/libraries/gnet-1.1.0-1.i386.rpm
rpm -Uvh /mnt/cdrom/libraries/gtkglarea-1.2.2-2.i386.rpm
rpm -Uvh /mnt/cdrom/libraries/gtkglarea-develop-1.2.2-2.i386.rpm

rpm -Uvh /mnt/cdrom/webots/webots-3.0.1-1.i386.rpm
```
3. You may need to use the `--nodeps` or `--force` options if the `rpm` command fails to install the packages.
4. Copy your personal `webots.key` file into the `/usr/local/webots/` directory where Webots was just installed.

1.3.2 Windows 95, 98, 2000, ME and NT

To install Webots on your Windows computer, perform the following steps:

1. Uninstall any previous release of Webots, if any, from the `Start` menu, `Control Panel`, `Add/Remove Programs`, or from the `Start` menu, `Programs`, `Cyberbotics`, `Uninstall Webots`.
2. Insert the Webots CD-ROM and open it.
3. Go to the `webots\windows` directory on the CD-ROM.
4. Double click on the `webots-3.0.1_setup.exe` file.
5. Follow the installation instructions.
6. Copy your personal `webots.key` file into the `C:\Program Files\Webots 3.0` directory where Webots was just installed.

In order to be able to compile controllers, you will need to install a C/C++ development environment. Dev-C++ is provided on the Webots CD-ROM. It is an Open Source Integrated Development Environment running on Windows and includes the GNU GCC compiler and utilities. To install it, unzip the `devcpp4.zip` file and follow the installation instructions. You may also install the update `devcpp401.zip` for a more up-to-date version of the software. Please note that Dev-C++ is Open Source software licensed under the GNU GPL. To learn more about this great program, visit the developers' website at `http://www.bloodshed.net`.

Here are some explanations about the installation procedure:

1.3.2.1 OpenGL on Windows

Webots uses the OpenGL graphics library for rendering the 3D scene. If you installed the hardware accelerated version of the `opengl32.dll` shipped with your 3D video card, you will get hardware acceleration for the rendering, hence simulation will run much faster. Webots has been tested on nVidia TNT2 32bits and on 3Dfx Voodoo Banshee graphics cards for hardware acceleration, but any other 3D video card with OpenGL support should provide you with accelerated 3D rendering in Webots.

1.3.2.2 Uninstallation

Before installing a new version of Webots, you must uninstall the old version (if any). You can do this either from the `Start` menu, `Control Panel`, `Add/Remove Programs`, or from the `Start` menu, `Programs`, `Cyberbotics`.

If any files are remaining in the `C:\Program Files\Webots 3.0`, double-check that you really want to delete them (you may want to backup your personal `webots.key` license file, the worlds or controllers you wrote). After making a copy of the files you want to preserve, delete the whole `Webots 3.0` directory.

Chapter 2

Upgrading from Webots 2.0

If you have already worked with Webots 2.0, your existing programs need to be modified for use with Webots 3.0:

- ○ The world files have to be edited with a text editor to match the file format of Webots 3.0.
- ○ The C controllers have to be slightly changed to meet the requirements of the new programming interface, including the use of GTK+ which replaces the GUI of Webots 2.0.

Warning: If you work on a Linux platform with Khepera robots using a gripper or if you would like to transfer your simulations to the real robot, we advise you to continue using version 2 of Webots. The virtual Khepera gripper will be available in a forthcoming version of Webots 3.

2.1 World

You first have to edit your Webots 2.0 world with a text editor to make it understandable by Webots 3.0. 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 2.0 nodes no longer exist: `Ball, Can, Ground, KheperaFeeder, Lamp, Wall, Alice, AliceIRCom, Khepera, KheperaK213, KheperaK6300, KheperaGripper, KheperaPanoramic, Supervisor`. They must be replaced by their new equivalents, as illustrated in table 2.1:

Nodes in Webots 2.0	Nodes in Webots 3.0	including (Webots 3.0)
`Ball`	`Solid`	`Sphere`
`Can`	`Solid`	`Cylinder`
`Ground`	`Solid`	`ElevationGrid`
`Lamp`	`Solid`	`Sphere`
`Wall`	`Solid`	`Extrusion`
`Alice`	`DifferentialWheels`	`DistanceSensor`
`AliceIRCom`	`Receiver`	
`Khepera`	`DifferentialWheels`	`DistanceSensor`
`KheperaK213`	`Camera`	
`KheperaK6300`	`Camera`	
`KheperaGripper`	not available	
`KheperaPanoramic`	not available	
`Supervisor`	`Supervisor`	

Table 2.1: Nodesequivalents between Webots 2 and Webots 3

2.2 Controller

2.2.1 Location

The controller program is still found in the `controllers` directory as 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.supervisor`), 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 2.0	Webots 3.0
`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 3

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

```
#include <robot.h>
#include <differential_wheels.h>
#include <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 2.0 and is well documented. You can find the GTK+ documentation (Tutorial and Reference Manual) on the Webots 3.0 CD-ROM, in books available in computer science libraries, and on the Internet (`http://www.gtk.org`).

2.3 Supervisor

The syntax of supervisor functions has changed a lot from Webots 2.0 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.

Chapter 3

Getting Started with Webots

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

1. A 3D virtual world containing one or more 3D virtual robots described by the `.wbt` file.

2. A C / C++ or Java program to control each robot, and optionally, a C/C++ supervisor program to control the simulation.

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 Chapters 5 and 6.

3.1 Running Webots

3.1.1 UNIX Systems

From a UNIX prompt, type `webots` to launch the simulator. You should see the world window appear on the screen (see figure 3.1).

3.1.2 Windows Systems

From Windows click the `start` button on the system bar, go to the `Programs|Cyberbotics` menu and click on `Webots 3.0`. You should see the world window appear on the screen (see figure 3.1).

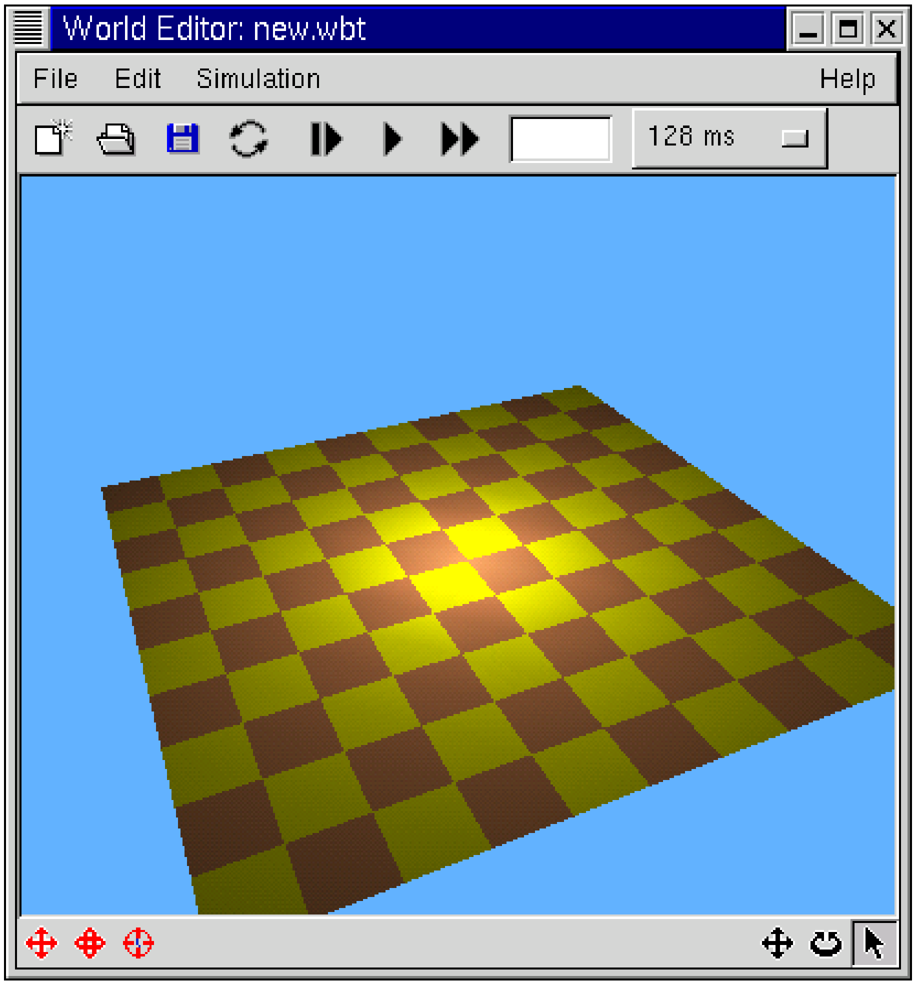


Figure 3.1: New world

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.

3.2.1 `File` menu and shortcuts

The `New` item opens a new default world representing a chessboard of 10 * 10 plates on a surface of 1 m * 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 computer port used for HyperGate networking.

  Checking the `Display sensor rays` check box displays the distance sensor rays of the robot(s) as red lines.

- `Files and paths`: The default `.wbt` world, the directory where it is, and the directory of the controllers are defined here.

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 selecting the pointer button and double-clicking on a solid in the world. A solid is a physical object in the world (see subsection 3.3.3.1).

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.

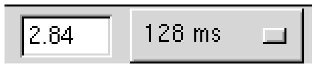


Figure 3.2: Speedometer

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.

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 let you visit our Web site.

3.2.5 Buttons navigating 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. Three buttons at the bottom of the window can be used to navigate in the 3D scene. 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.

- `Translate viewpoint`: To translate the camera in the x and y directions, you can click the left button of the mouse and then drag the mouse on the scene. Using the right button of the mouse zooms in and out of the scene.

- `Rotate viewpoint`: To rotate the camera around the x and y axis, you have to click on the button and then:

  1. 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.
  2. if you click outside of any solids and drag the mouse the rotation will be centered around the origin of the world coordinate system.

  Using the right button of the mouse translates the scene.

- `Zoom / Tilt viewpoint`: you must first click on the scene and then:

  1. if you drag the mouse vertically, the camera will zoom in or out.
  2. if you drag the mouse horizontally, the camera will rotate around its z axis.

Using the right button of the mouse rotates the scene.

3.2.6 Buttons for moving a solid object

The two following buttons allow you to select and move solid objects in the world:

- ✥ This move button allows you to translate a solid object on the ground (xz plan) using a standard drag'n'drop movement.

- ↻ This turn button allows you to rotate solid objects around the vertical axis (y): click on this button to enter the turn mode, then click on an object to select it and then click somewhere else and drag the mouse pointer to rotate the selected solid object around the y axis.

3.2.7 Pointer button

↖ Clicking on the pointer button allows you to select solid objects. Clicking on the pointer button and then on a robot enables the choice of `Robot View` in the simulation menu. Clicking on the pointer button and then double-clicking on a solid 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

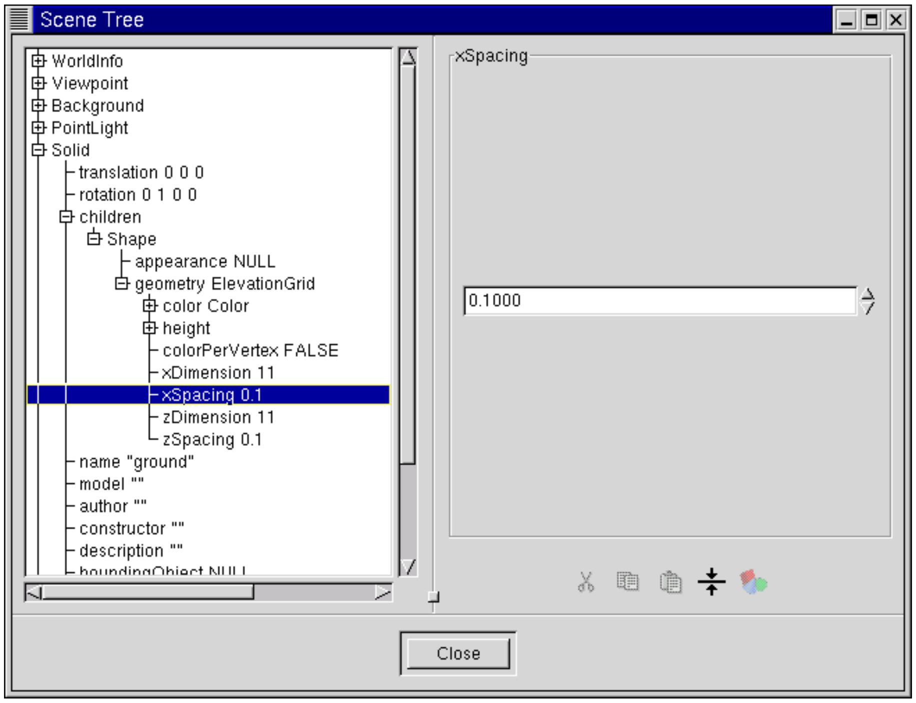
As seen in the previous section, to access to the Scene Tree Window you can either choose `Scene Tree Window` in the `Edit` menu, or click on the pointer button and double-click on a solid object.

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.



```
⊞ WorldInfo
⊞ Viewpoint
⊞ Background
⊞ PointLight
⊟ Solid
    ├ translation 0 0 0
    ├ rotation 0 1 0 0
    ⊟ children
        ⊟ Shape
            ├ appearance NULL
            ⊟ geometry ElevationGrid
                ⊞ color Color
                ⊞ height
                ├ colorPerVertex FALSE
                ├ xDimension 11
                ├ xSpacing 0.1
                ├ zDimension 11
                └ zSpacing 0.1
    ├ name "ground"
    ├ model ""
    ├ author ""
    ├ constructor ""
    ├ description ""
    ├ boundingObject NULL
```

Figure 3.3: Scene Tree Window

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`
- `Copy`
- `Paste after`

  These three buttons let you cut, copy and paste nodes but not fields. However, you can't perform these operations with 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.

- `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.

3.3.2 VRML nodes

A number of VRML 2.0 nodes are partially or completely supported in Webots 3.0.

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: `http://www.vrml.org`.

The VRML nodes supported in Webots are the following:

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

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

The Reference Manual gives a more comprefensive 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.

3.3.3.1 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.

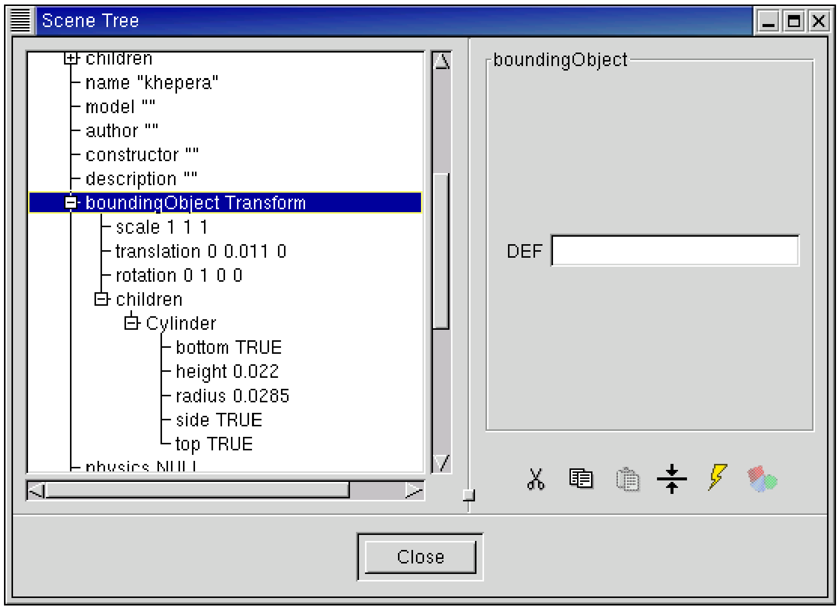
  A description of the fields of the `Solid` node is given below.

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.
- `consructor`: 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 not used in this version of Webots. It will be used in a future version to modelize the physical properties of a solid.
- `joint`: idem `physics` field.

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 `boundin-gObject` field of the `DifferentialWheels` node. The bounding object is a cylinder which has been transformed. See figure 3.4.



```
Scene Tree
 children
 name "khepera"
 model ""
 author ""
 constructor ""
 description ""
 boundingObject Transform
    scale 1 1 1
    translation 0 0.011 0
    rotation 0 1 0 0
    children
       Cylinder
          bottom TRUE
          height 0.022
          radius 0.0285
          side TRUE
          top TRUE
 physics NULL
```

Figure 3.4: The bounding box of the Khepera robot

3.3.3.2 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.
- `synchronisation`: if the value is `TRUE`, 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.

Note: $[J] = [V].[A].[s]$

$[J] = [V].[A.h]/3600$

- `cpuConsumption`: consumption of the cpu in Watts.
- `motorConsumption`: consumption of the motor in Watts.
- `axleLength`: distance between the two wheels.
- `wheelRadius`: radius of the wheels. Both wheels must have the same radius.
- `targetSpeed`: current target speed to be reached by the wheels; two values in rad/s: the first value is the target speed of the left wheel and the second value is the target speed of the right wheel.
- `maxSpeed`: maximum speed of the wheels; one value in rad/s.
- `maxAcceleration`: maximum acceleration the wheels can have, in rad/s^2.
- `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.

3.3.3.3 The DistanceSensor node

The `DistanceSensor` node is used to model sonar sensors, infra-red sensors and laser range finders. It inherits from the `Solid` node. The `DistanceSensor` node includes two specific fields:

- `fieldOfView`: It corresponds to the angle width of the sensor. Its values range from 0 to π radians.
- `lookupTable`: Example of a lookup table:

  Let us consider an infra-red sensor. The noise on the reading is 10 percent. For an obstacle made of a given material and colour and for a given ambient light, the response of the sensor is as shown in figure 3.5

  The values of the `lookupTable` will be:

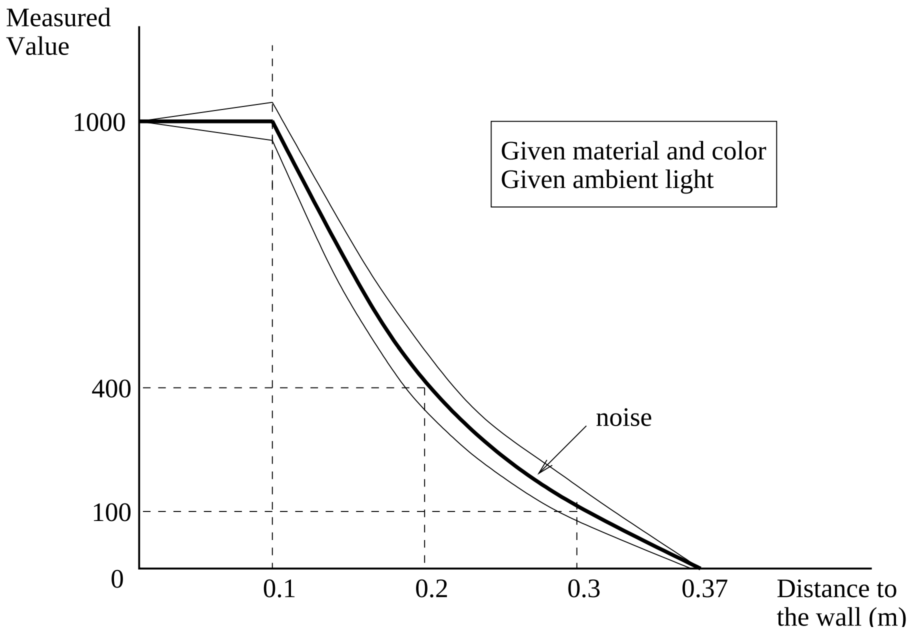


Figure 3.5: Measurements of the light reflected by an obstacle

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

- `type`: type of sensor: “infra-red”, “sonar” or “laser”.

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 `General` panel.

3.3.3.4 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 π radians.
- `width`: width of the image in pixels.
- `height`: height of the image in pixels.
- `type`: type of the camera: “color” (“black and white” doesn’t exist yet).

3.3.3.5 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 to recharge itself. A charger is not a 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.

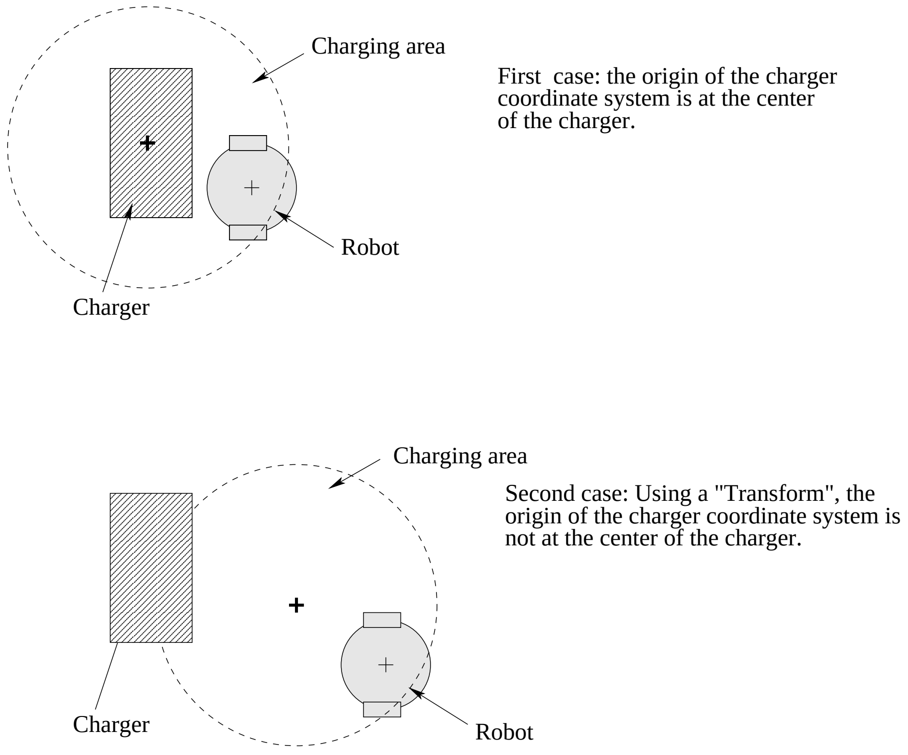

Charging area
First case: the origin of the charger coordinate system is at the center of the charger.
Robot
Charger
Charging area
Second case: Using a "Transform", the origin of the charger coordinate system is not at the center of the charger.
Charger
Robot

Figure 3.6: The sensitive area of a charger

3.3.3.6 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: “ir” 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 infrared emitter or a frequency for a radio emitter. The receiver must use the same channel to receive the emitted signals.
- `baudRate`: the baudRate value is the communication speed expressed in number of bits per second.
- `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.

3.3.3.7 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.

The fields specific to the `Receiver` node are:

- `type`: type of the received signals: “ir” or “radio”.
- `channel`: channel of reception. The value is an identification number for an infrared receiver or a frequency for a radio receiver. The receiver 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 new data is received and if old data were not yet read, the old data are lost.

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

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

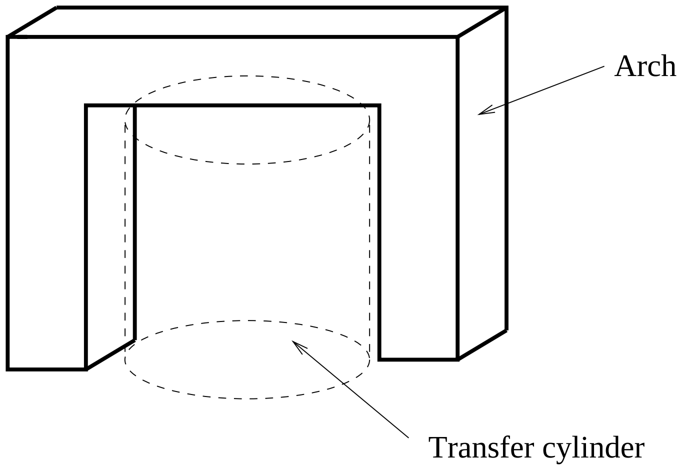


Figure 3.7: An example of a hypergate

3.3.3.9 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.

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 simple modifications using a text editor.

Chapter 4

Tutorial: Modelling and simulating your robots

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 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 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.

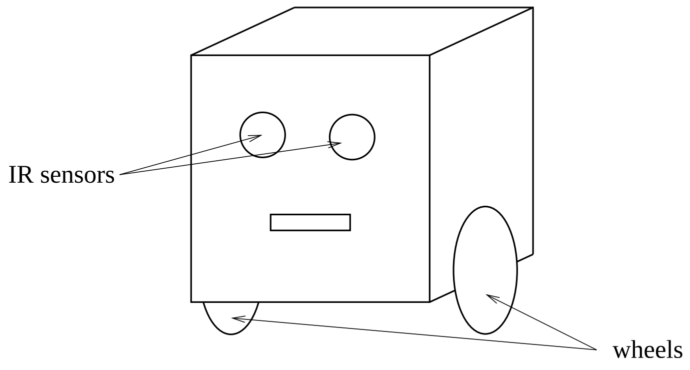
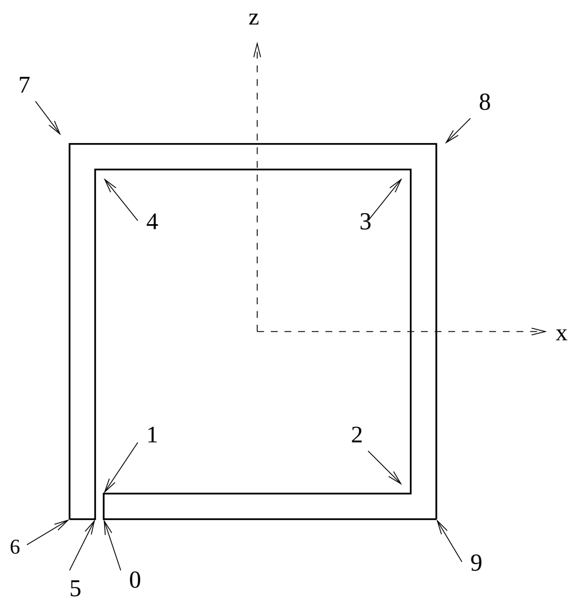


Figure 4.1: The *kiki* robot

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.



(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: Coordinates of the corners of the wall

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:

- Select the `PointLight` node, and click on the + just in front of it. You can now see the different fields of `PointLight`. 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*.

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

- 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.

- The scene is now better lit.

Secondly, let us create the wall:

- Select the last node (`Solid`), and click on the *insert after* button.

- Choose a `Solid` node.
- Fill the text fields with your desired text, e.g., “wall” for the name.
- Select the `children` field and *Insert after* a `Shape` node.
- *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.
- Now insert an `Extrusion` node in the `geometry` field of the `Shape`.
- Set the wall corner coordinates in the `crossSection` field and set `convex` to `FALSE`.
- In the `spine` field, write that the wall ranges between 0 and 0.1 along the y axis.
- As we want our robot to detect the wall, we have to fill in the `boundingObject` field. The bounding box 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`.
- Close the tree editor, save your file and look at the result.

Figures 4.3 and 4.4 show the wall in the tree editor and in the world editor.

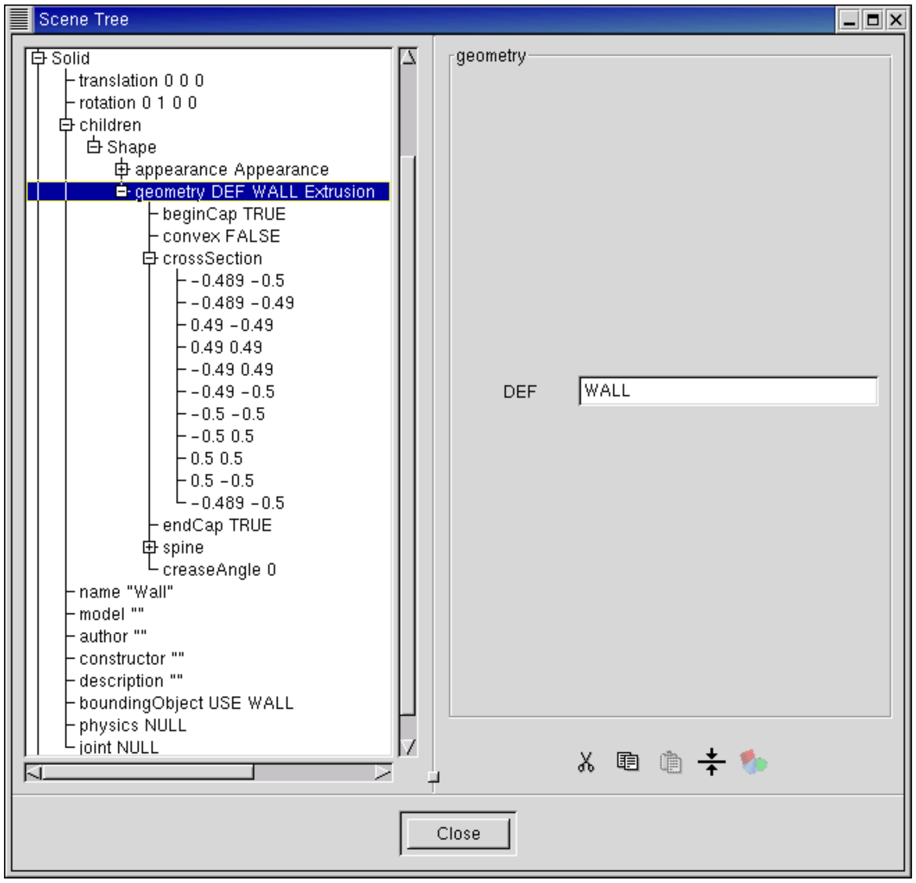
4.1.2 Robot

The aim of this subsection is to model the *kiki* robot. This robot is made 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.

Figure 4.5 shows the origin and the axis of the coordinate system of the robot and its dimensions.

To model the body of the robot:

- Open the tree editor.
- Select the last `Solid` node.
- *Insert after* a `DifferentialWheels` node, give and it the name “kiki”.
- 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. The dimension of the box is [0.08 0.08 0.08]. Now add some `translation` values [0 0.06 0] in the `Transform` node (see figure 4.6).



```
Solid
  translation 0 0 0
  rotation 0 1 0 0
  children
    Shape
      appearance Appearance
      geometry DEF WALL Extrusion
        beginCap TRUE
        convex FALSE
        crossSection
          -0.489 -0.5
          -0.489 -0.49
          0.49 -0.49
          0.49 0.49
          -0.49 0.49
          -0.49 -0.5
          -0.5 -0.5
          -0.5 0.5
          0.5 0.5
          0.5 -0.5
          -0.489 -0.5
        endCap TRUE
        spine
        creaseAngle 0
  name "Wall"
  model ""
  author ""
  constructor ""
  description ""
  boundingObject USE WALL
  physics NULL
  joint NULL
```

DEF WALL

Figure 4.3: The wall in the tree editor

To model the left wheel of the robot:

- 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.

- The axis of rotation of the wheel is x. Moreover, a wheel is made of a `Cylinder`, with a rotation 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.

- 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`. Add the `rotation` [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.

- 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].

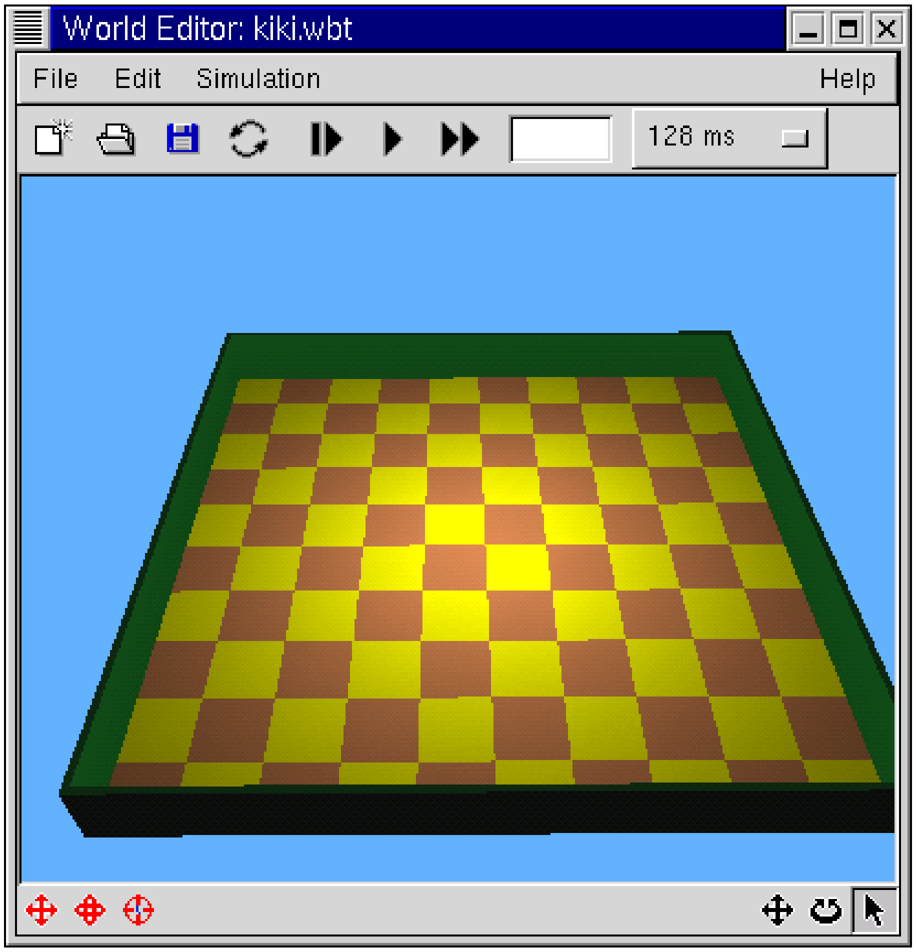


Figure 4.4: The wall in the world editor

- Give a `DEF` name to your `Tranform`: 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.

- 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:

- 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].

- In `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.

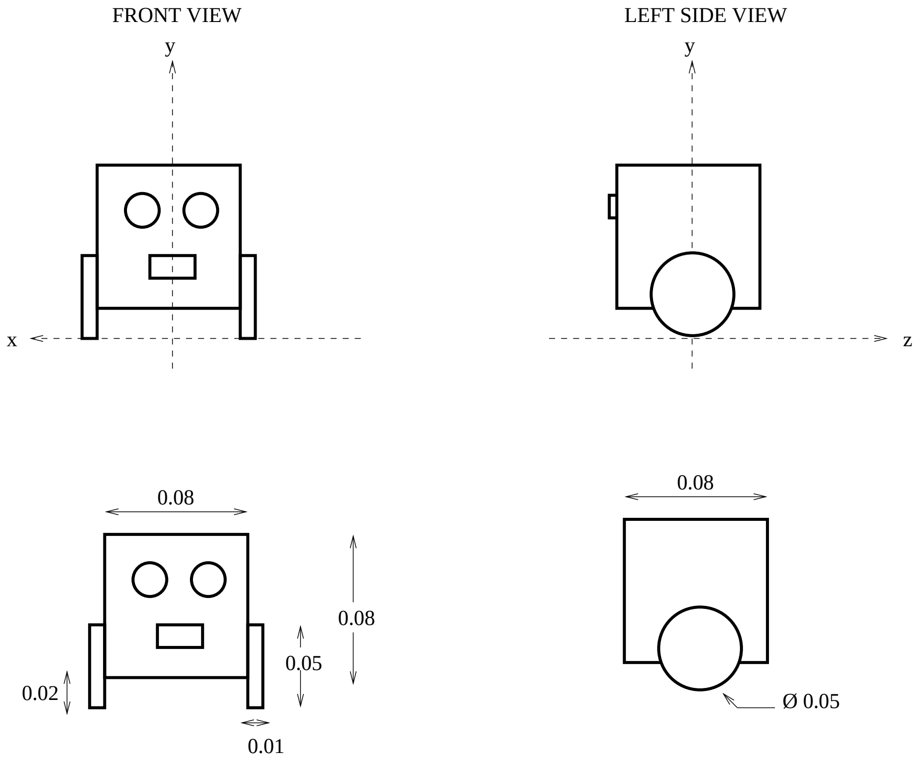
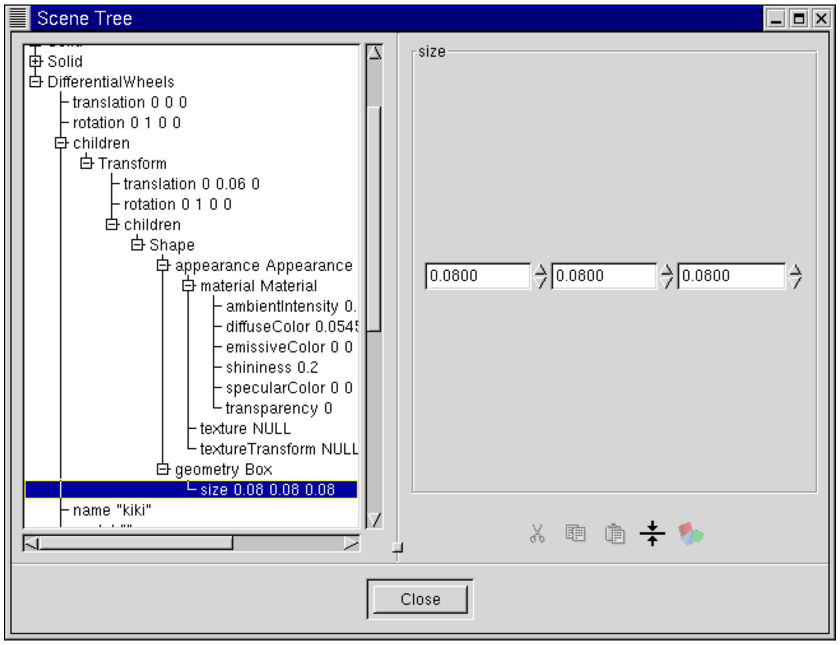


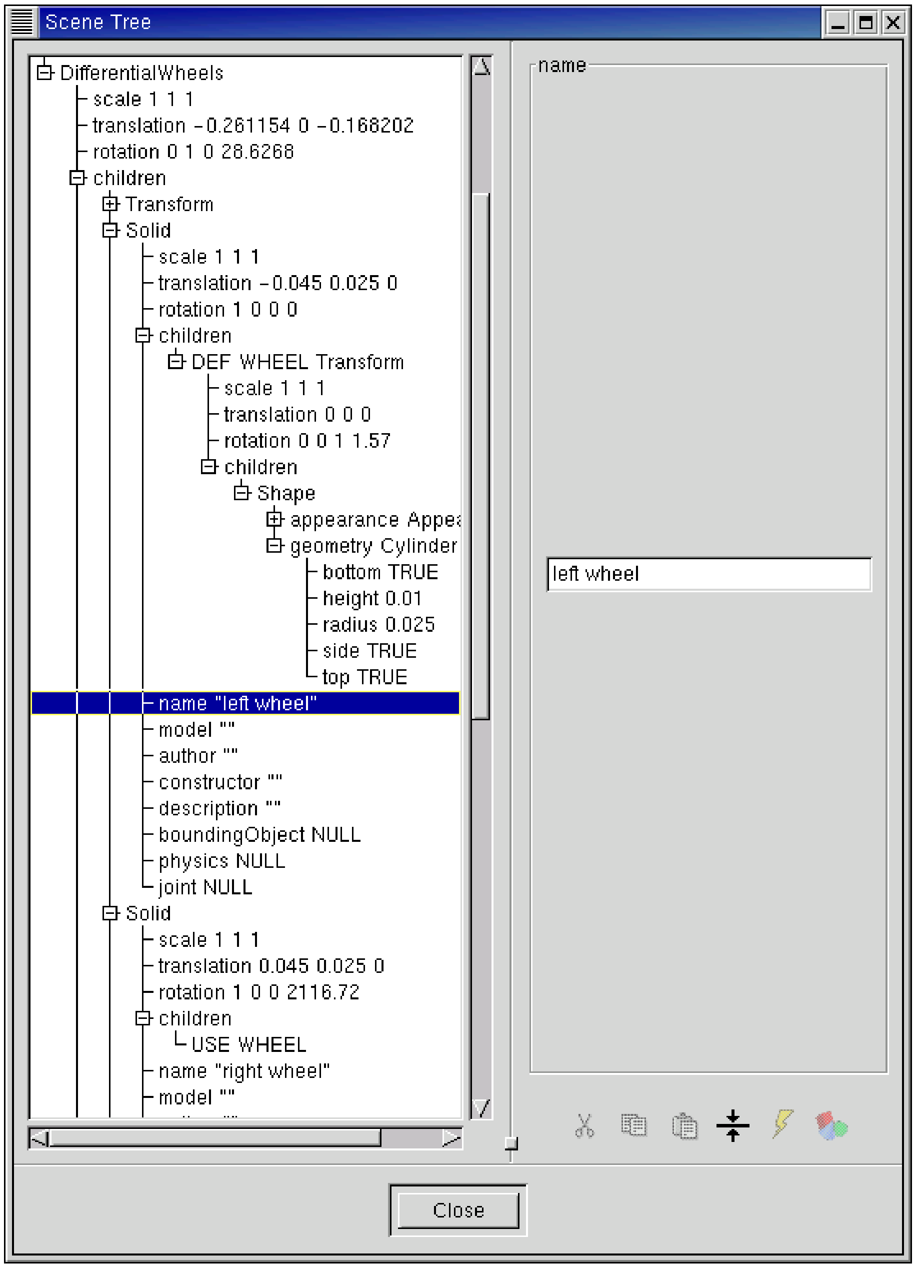
Figure 4.5: Coordinate system and dimensions of the *kiki* robot



```
Solid
DifferentialWheels
  translation 0 0 0
  rotation 0 1 0 0
  children
    Transform
      translation 0 0.06 0
      rotation 0 1 0 0
      children
        Shape
          appearance Appearance
            material Material
              ambientIntensity 0.
              diffuseColor 0.0545
              emissiveColor 0 0
              shininess 0.2
              specularColor 0 0
              transparency 0
            texture NULL
            textureTransform NULL
          geometry Box
            size 0.08 0.08 0.08
  name "kiki"
```

size: 0.0800 0.0800 0.0800

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



```
DifferentialWheels
  scale 1 1 1
  translation -0.261154 0 -0.168202
  rotation 0 1 0 28.6268
  children
    Transform
    Solid
      scale 1 1 1
      translation -0.045 0.025 0
      rotation 1 0 0 0
      children
        DEF WHEEL Transform
          scale 1 1 1
          translation 0 0 0
          rotation 0 0 1 1.57
          children
            Shape
              appearance Appea
              geometry Cylinder
                bottom TRUE
                height 0.01
                radius 0.025
                side TRUE
                top TRUE
      name "left wheel"
      model ""
      author ""
      constructor ""
      description ""
      boundingObject NULL
      physics NULL
      joint NULL
    Solid
      scale 1 1 1
      translation 0.045 0.025 0
      rotation 1 0 0 2116.72
      children
        USE WHEEL
      name "right wheel"
      model ""
```

```
name

left wheel
```

Figure 4.7: Wheels of the *kiki* robot

The robot and its two wheels are shown in figures 4.7 and 4.8.

To model the two infra-red sensors:

The two IR 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.

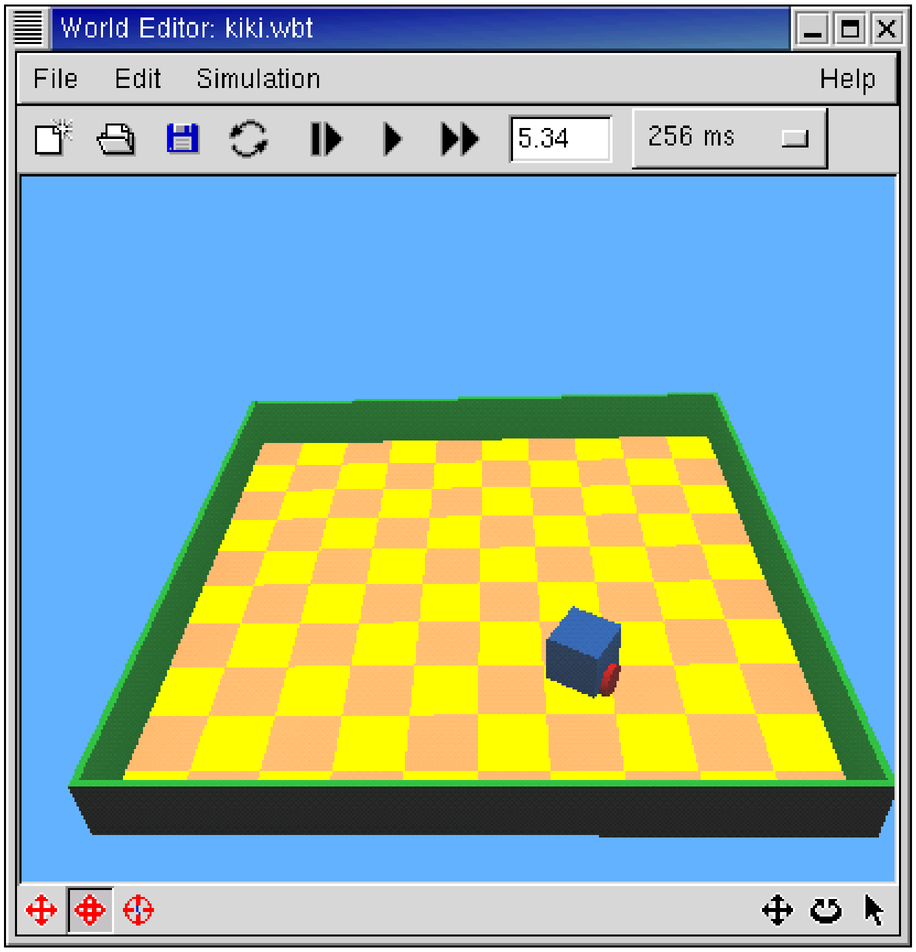


Figure 4.8: Body and wheels of the *kiki* robot

- In the `children` of the `DifferentialWheels` node, *insert after* a `DistanceSensor` node.

- Type the name “ir0”. It will be used by the controller program.

- You must transform a `Cylinder` to put it on the face of the robot. In the `children` 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.

- In the `children` of the `Tranform` node, *insert after* a `Shape`. Choose an appearance and *insert* a `Cylinder` in the `geometry` field. Type 0.004 for the height and 0.08 for the radius.

- Add a rotation in the `Tranform` node: [0 0 1 1.57] to adjust the orientation of the cylinder.

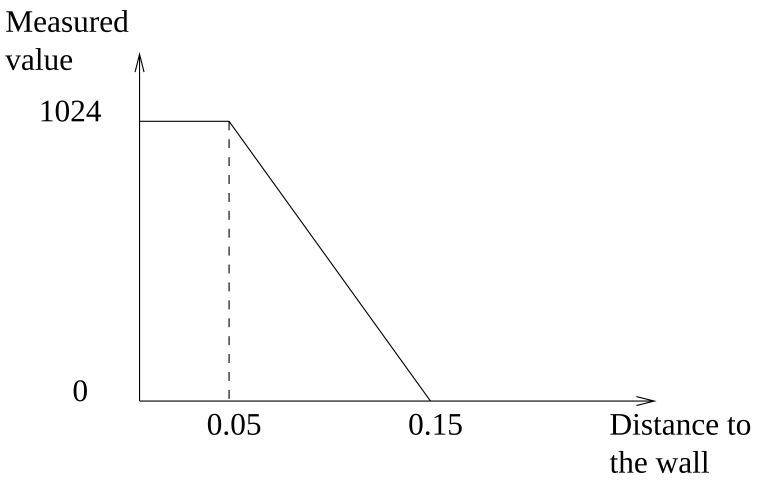


Figure 4.9: Light measurements of the *kiki* robot sensors

- In the `DistanceSensor` node, add a `translation` to position the sensor and its ray: [0.02 0.08 -0.042]. In the `File` menu, `Preferences` item, `General`, check the `Display sensor rays` box. In order to have the ray directed towards the front of the robot, you must add a `rotation`: [0 1 0 1.57].
- 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 ]
```

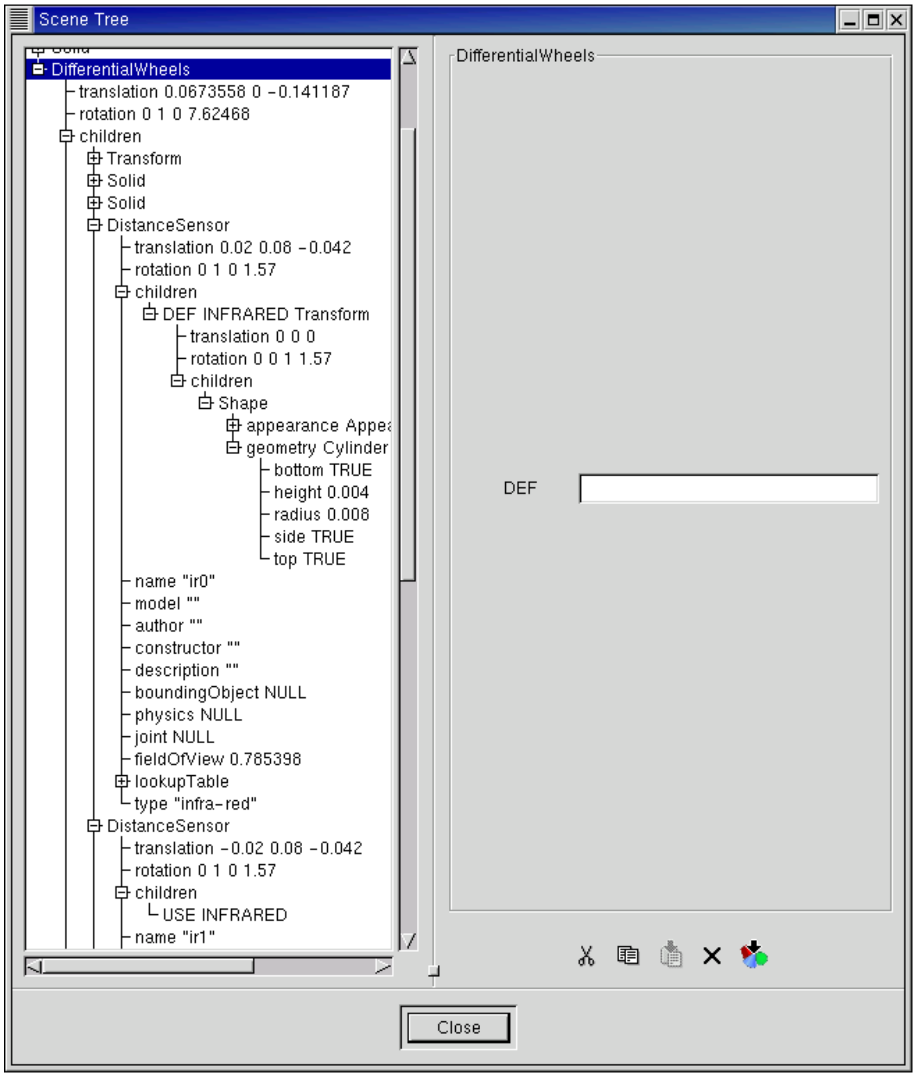
- To model the second IR sensor, select the `DistanceSensor` node and *insert after* a new `DistanceSensor` node. Type the name “ir1”. Add the `translation` [-0.02 0.08 -0.042] and the `rotation`: [0 1 0 1.57]. In the `children`, *insert after* `USE INFRARED`. In the `lookupTable` field, type the same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

The robot and its two sensors are shown in figures 4.10 and 4.11.

Note on the textures in Webots: a texture can only be mapped on an IndexedFaceSet shape. The `texCoord` and `texCoordIndex` entries must be filled. The image used as a texture must be a `.png` or a `.jpg` file, and its size must be 2^nx2^n pixels (for example 8x8, 16x16, 32x32, 64x64, 128x128 pixels). Transparency images are not allowed in Webots.

To paste a texture on the face of the robot:

- Select the last `DistanceSensor` node and *insert after* a `Shape` node.



```
DifferentialWheels
  translation 0.0673558 0 -0.141187
  rotation 0 1 0 7.62468
  children
    Transform
    Solid
    Solid
    DistanceSensor
      translation 0.02 0.08 -0.042
      rotation 0 1 0 1.57
      children
        DEF INFRARED Transform
          translation 0 0 0
          rotation 0 0 1 1.57
          children
            Shape
              appearance Appea
              geometry Cylinder
                bottom TRUE
                height 0.004
                radius 0.008
                side TRUE
                top TRUE
      name "ir0"
      model ""
      author ""
      constructor ""
      description ""
      boundingObject NULL
      physics NULL
      joint NULL
      fieldOfView 0.785398
      lookupTable
      type "infra-red"
    DistanceSensor
      translation -0.02 0.08 -0.042
      rotation 0 1 0 1.57
      children
        USE INFRARED
      name "ir1"
```

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

◦ In the `appearance` field, *insert* an `Appearance` node. In the `texture` field of this node, *insert* an `ImageTexture` node with the URL `''kiki/kiki.png''`.

◦ 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,
 -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.

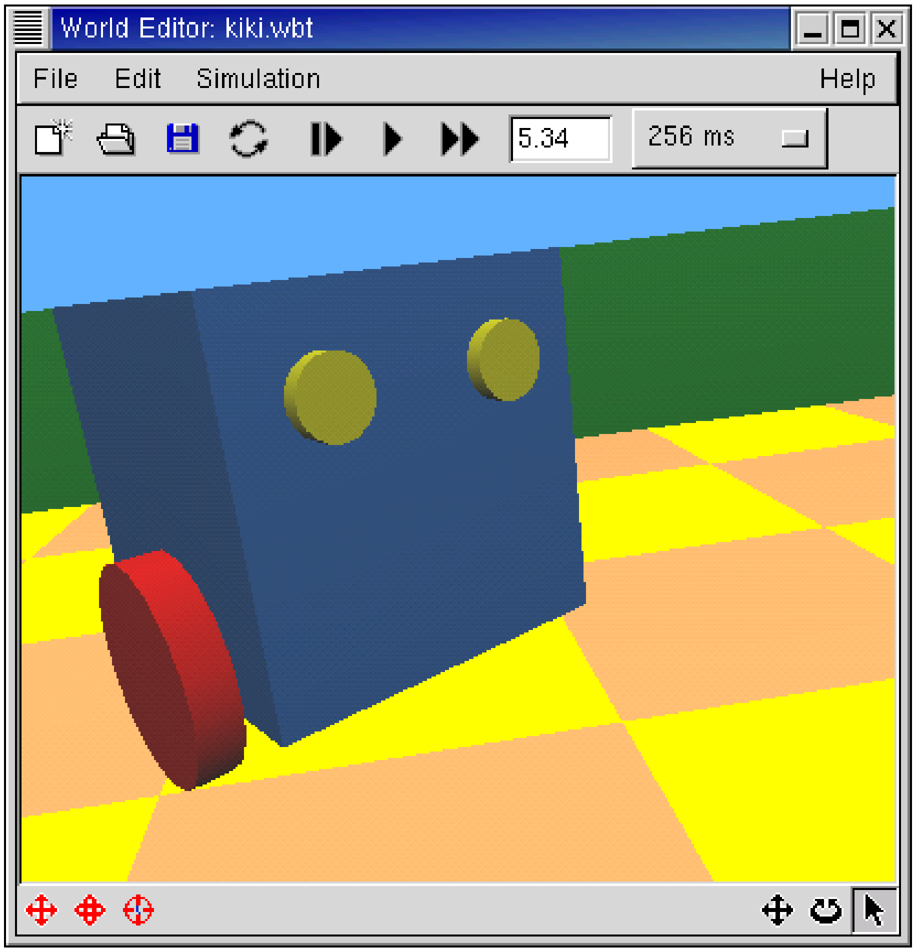


Figure 4.11: The *kiki* robot and its sensors

- In the `texCoord` field, *insert* a `TextureCoordinate` node. In the `point` field, enter the coordinates of the texture:

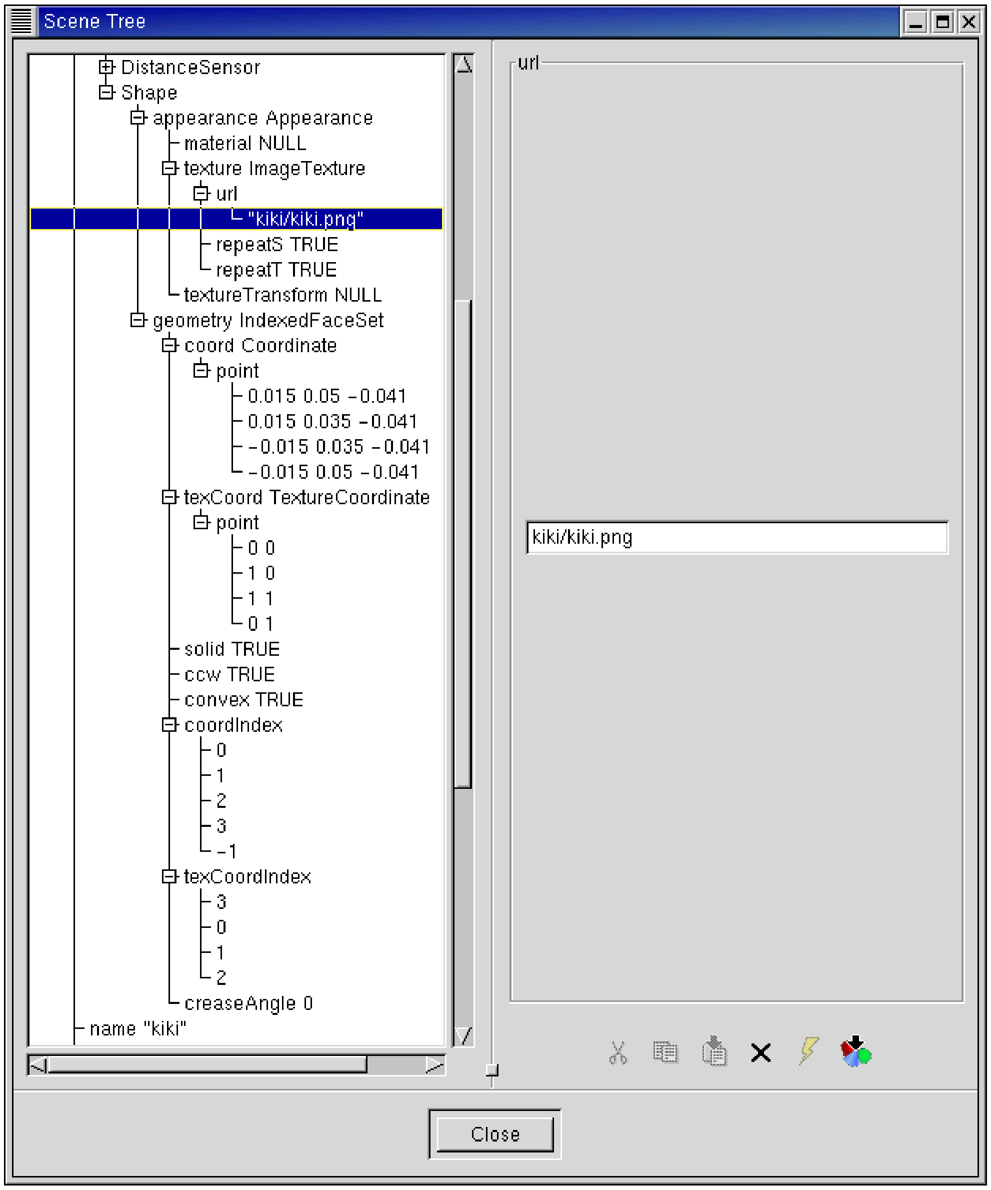
```
[ 0  0
  1  0
  1  1
  0  1 ]
```

  and in the `texCoordIndex` field, type 3, 0, 1, 2.

- The texture values are shown in figure 4.12.

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

- In the `controller` field, type the name “simple”. It will be used by the controller program.



```
DistanceSensor
Shape
  appearance Appearance
    material NULL
    texture ImageTexture
      url
        "kiki/kiki.png"
      repeatS TRUE
      repeatT TRUE
    textureTransform NULL
  geometry IndexedFaceSet
    coord Coordinate
      point
        0.015 0.05 -0.041
        0.015 0.035 -0.041
        -0.015 0.035 -0.041
        -0.015 0.05 -0.041
    texCoord TextureCoordinate
      point
        0 0
        1 0
        1 1
        0 1
    solid TRUE
    ccw TRUE
    convex TRUE
    coordIndex
      0
      1
      2
      3
      -1
    texCoordIndex
      3
      0
      1
      2
    creaseAngle 0
name "kiki"
```

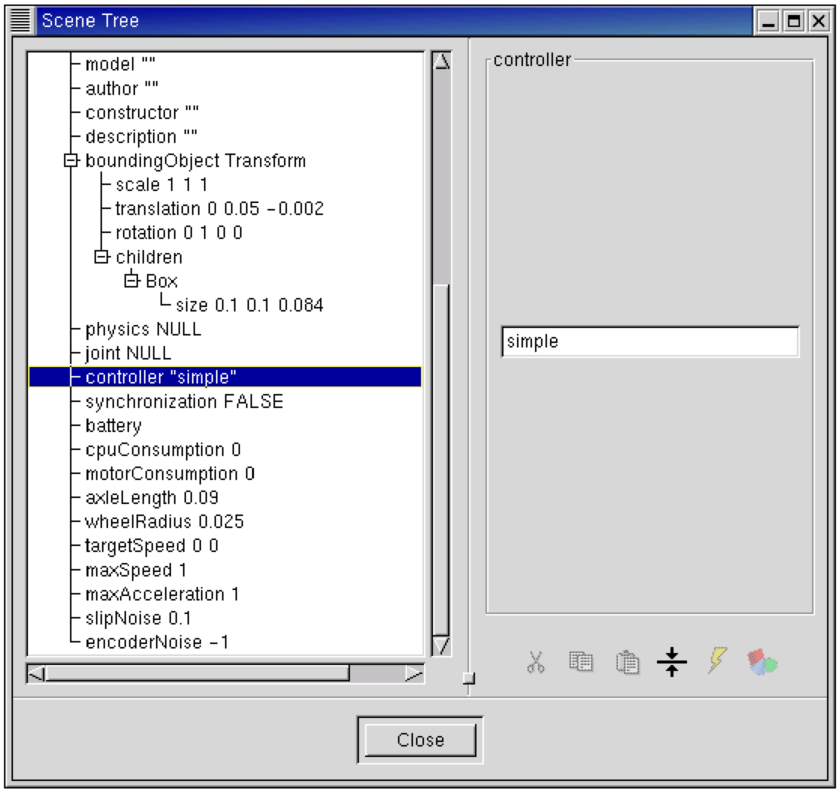
url: kiki/kiki.png

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

- 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` [0 0.05 -0.002] and a `Box` node in its `children`. The dimension of the `Box` is [0.1 0.1 0.084].

- In the `axleLength` field, enter the length of the axle between the two wheels: 0.09 (according to figure 4.5).

- In the `wheelRadius` field, enter the radius of the wheels: 0.025.



```
model ""
author ""
constructor ""
description ""
boundingObject Transform
  scale 1 1 1
  translation 0 0.05 -0.002
  rotation 0 1 0 0
  children
    Box
      size 0.1 0.1 0.084
physics NULL
joint NULL
controller "simple"
synchronization FALSE
battery
cpuConsumption 0
motorConsumption 0
axleLength 0.09
wheelRadius 0.025
targetSpeed 0 0
maxSpeed 1
maxAcceleration 1
slipNoise 0.1
encoderNoise -1
```

simple

Figure 4.13: The other fields of the DifferentialWheels node

- Values for other fields are shown in figure 4.13, and the finished robot in its world is shown in figure 4.14.

The file `kiki.wbt` is included on the CD-Rom.

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, such that *kiki* avoids the obstacles.

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

```
#include <robot.h>
#include <differential_wheels.h>
#include <distance_sensor.h>
#define SPEED 100
static device_tag ir0,ir1;
void reset(void) {
  ir0 = robot_get_device("ir0");
```

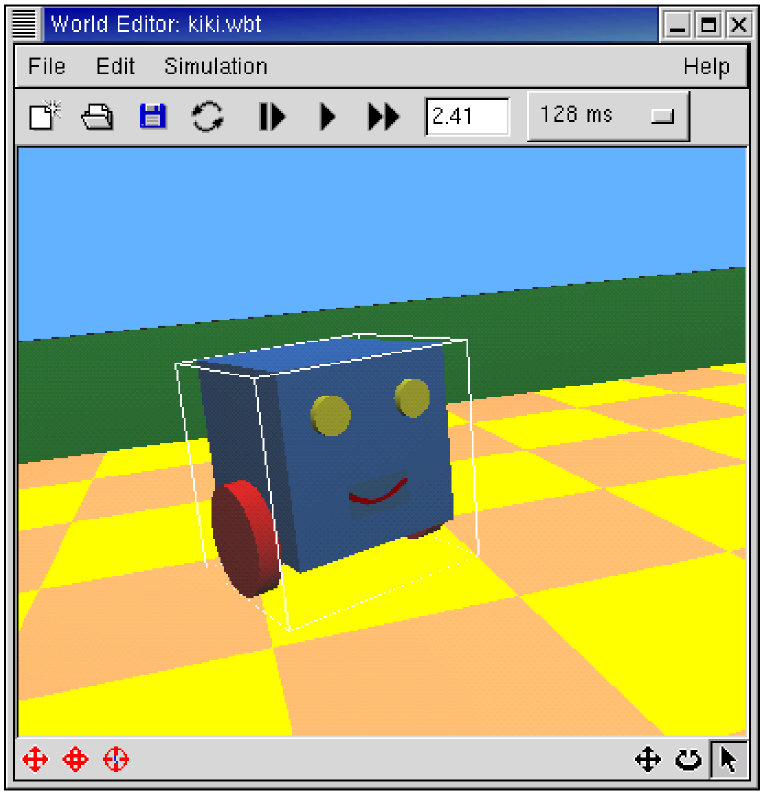


Figure 4.14: The *kiki* robot in its world

```
  ir1 = robot_get_device("ir1");
  //g_print("ir0=%d ir1=%d\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!  */
    ir0_value = distance_sensor_get_value(ir0);
    ir1_value = distance_sensor_get_value(ir1);
    if (ir1_value>200) {
0       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;
  }
```

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 file `kiki_camera.wbt` is included on the CD-Rom, if you need any help.

A controller program for this robot, called `camera` is also included on the CD-Rom.

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 file `Pioneer2.wbt` and its controller directory included on the CD-Rom.

4.3.1 Environment

The environment consists of:

- ◦ a chessboard : a `Solid` node with an `ElevationGrid` node.
- ◦ a wall around the chessboard : a `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.

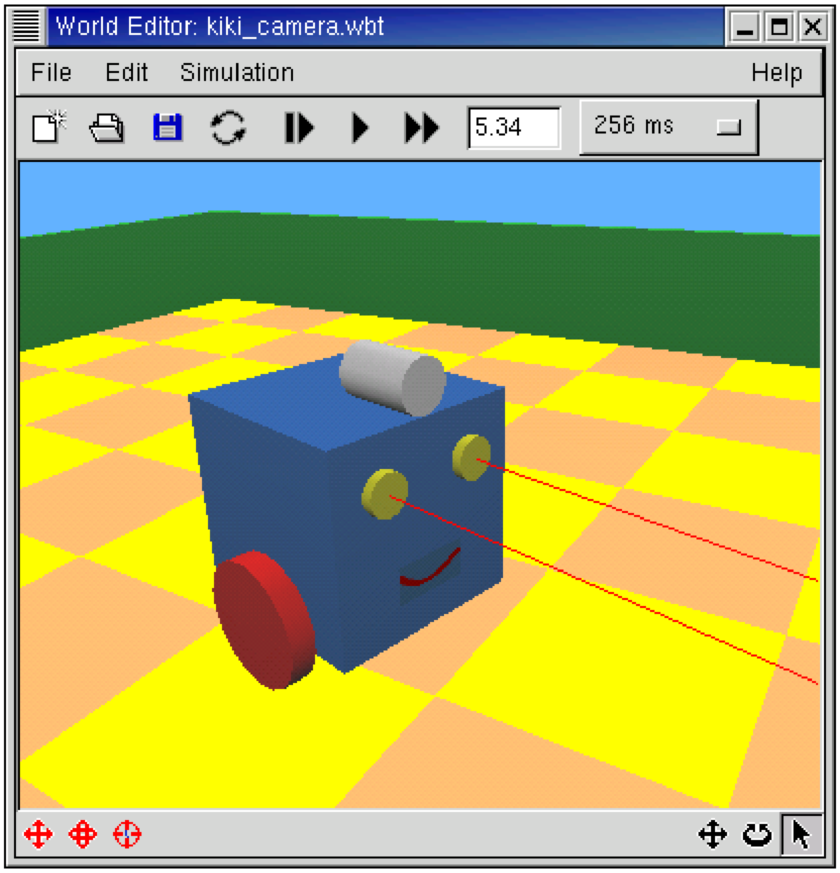


Figure 4.15: The *kiki* robot with a camera

4.3.2 Robot with 16 sonars

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

- the body: an `Extrusion` node.
- a top plate: an `Extrusion` node.
- two wheels: two `Cylinder` nodes.
- a rear wheel: a `Cylinder` node.
- front an rear sensor supports: two `Extrusion` nodes.
- sixteen sonars: sixteen `DistanceSensor` nodes.

Figure 4.17 describes the *Pioneer 2 DX* robot.

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

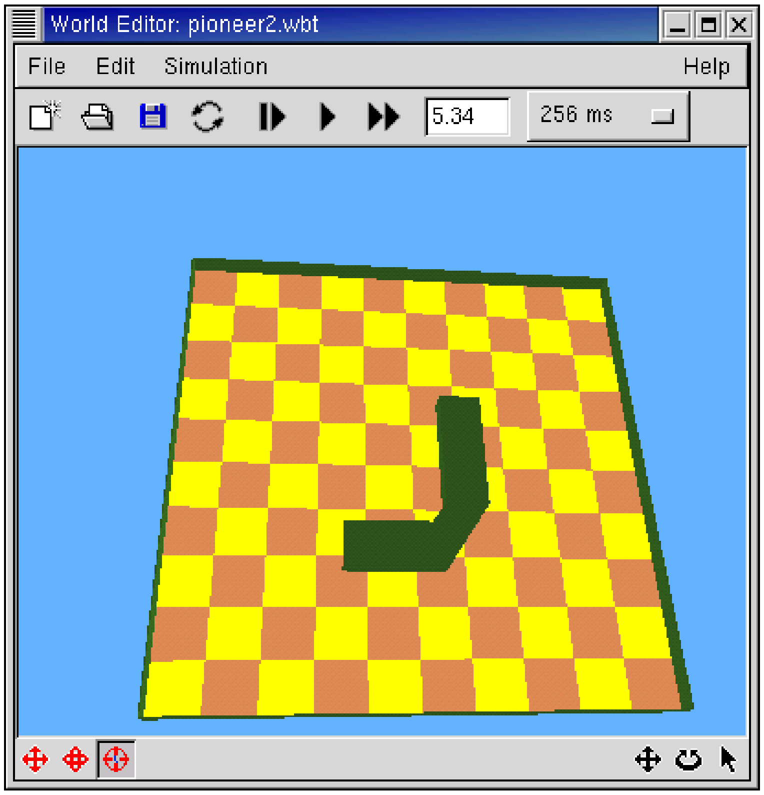


Figure 4.16: The walls of the *Pioneer 2* robot world

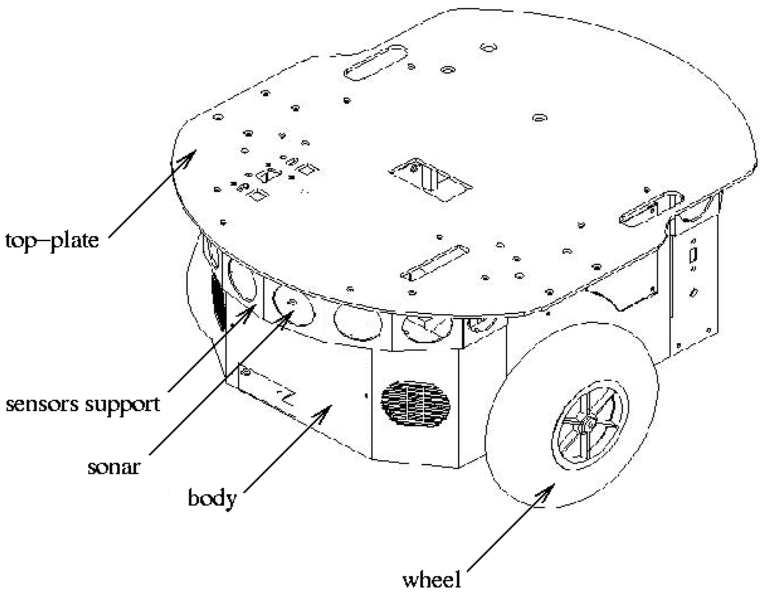
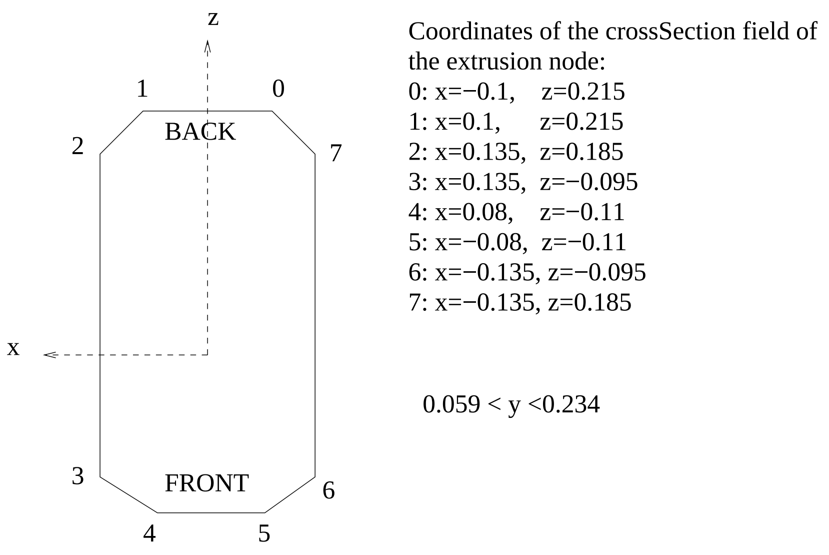


Figure 4.17: The *Pioneer 2 DX* robot

- for the body: a `Shape` node with a `geometry Extrusion`. See figure 4.18 for the `Extrusion` coordinates.



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

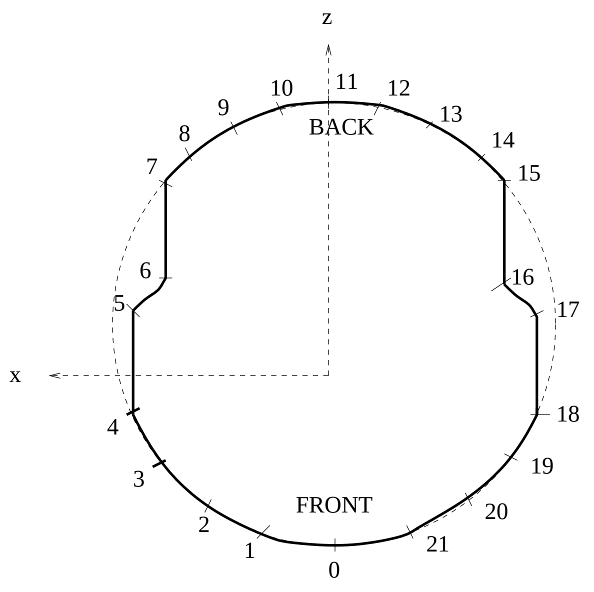
0.059 < y <0.234

Figure 4.18: Body of the *Pioneer2* robot

- for the top plate: a `Shape` node with a `geometry Extrusion`. See figure 4.19 for the `Extrusion` coordinates.
- for the two wheels: two `Solid` nodes. Each `Solid` node children contains a `Transform` node, within which is a `Shape` node with a `geometry Cylinder`. Each `Solid` node has a name: "left wheel", "right wheel". See figure 4.20 for the wheel dimensions.
- for the rear wheel: a `Transform` node containing a `Shape` node with a `geometry Cylinder`, as shown in figure 4.21.
- for the sonar supports: two `Shape` nodes with a `geometry Extrusion`. See figure 4.22 for the `Extrusion` coordinates.
- for the 16 sonars: 16 `DistanceSensor` nodes. Each `DistanceSensor` node contains a `Transform` node. The `Transform` node has a `Shape` node containing a `geometry Cylinder`. See figure 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 robot (see figure 4.23). Figure 4.24 shows the angles between the sonars and the initial position of the `DEF SONAR Transform`. 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.

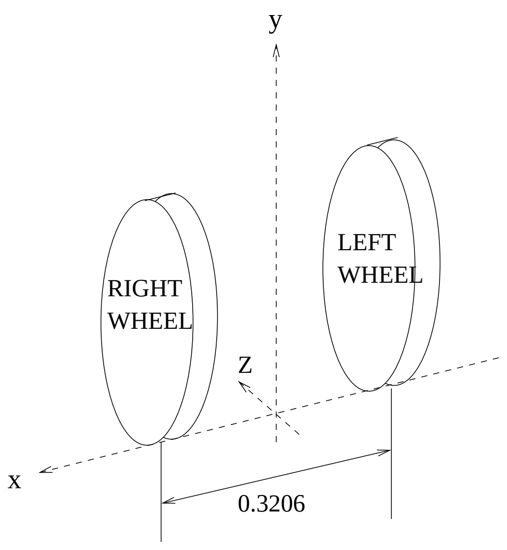


Coordinates of the crossSection field of the Extrusion node:

0:	x=0	z=−0.174
1:	x=−0.056	z=−0.166
2:	x=−0.107	z=−0.145
3:	x=−0.155	z=−0.112
4:	x=−0.190	z=−0.064
5:	x=−0.190	z=0.074
6:	x=−0.160	z=0.096
7:	x=−0.160	z=0.151
8:	x=−0.155	z=0.2
9:	x=−0.107	z=0.236
10:	x=−0.056	z=0.256
11:	x=0	z=0.264
12:	x=0.056	z=0.256
13:	x=0.107	z=0.236
14:	x=0.155	z=0.2
15:	x=0.160	z=0.151
16:	x=0.160	z=0.096
17:	x=0.190	z=0.074
18:	x=0.190	z=−0.064
19:	x=0.155	z=−0.112
20:	x=0.107	z=−0.145
21:	x=0.056	z=−0.166

$0.234 < y < 0.24$

Figure 4.19: Top plate of the *Pioneer 2* robot



Radius of the wheels: 0.0825
Depth of the wheels: 0.037

Figure 4.20: Wheels of the *Pioneer 2* robot

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.

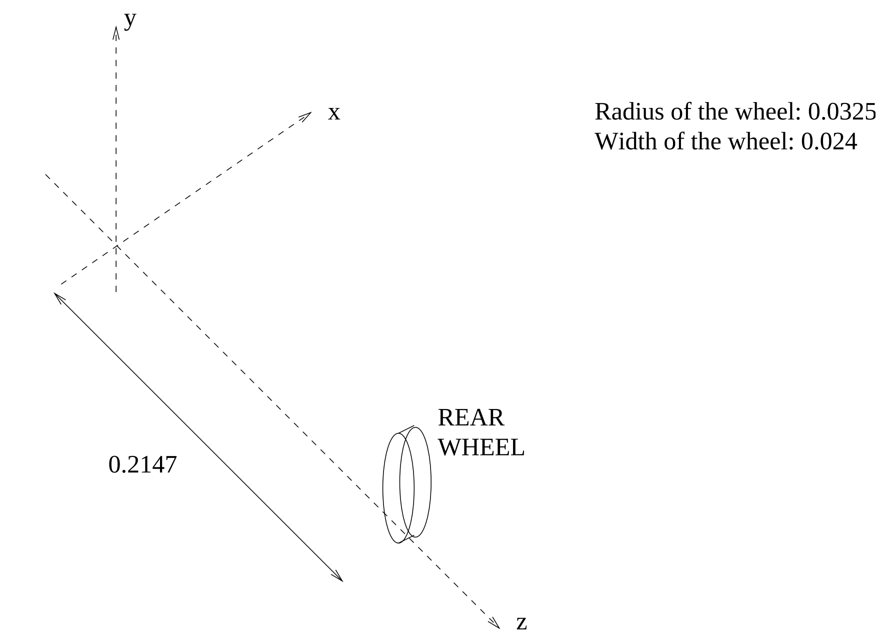
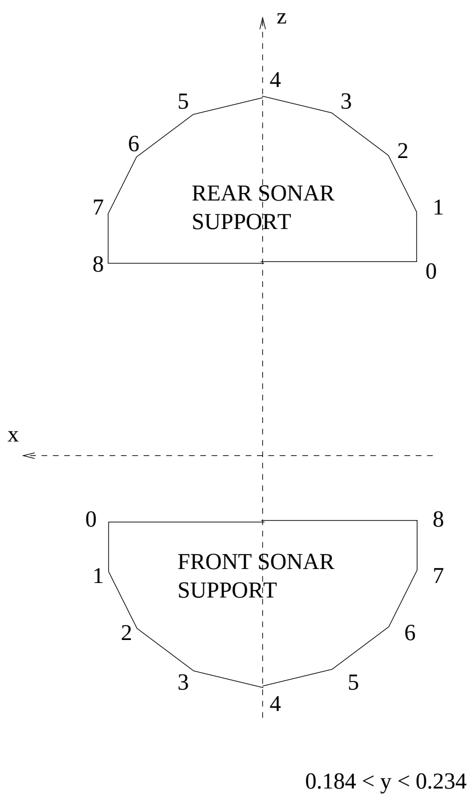


Figure 4.21: Rear wheel of the *Pioneer 2* robot



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 *Pioneer2* robot

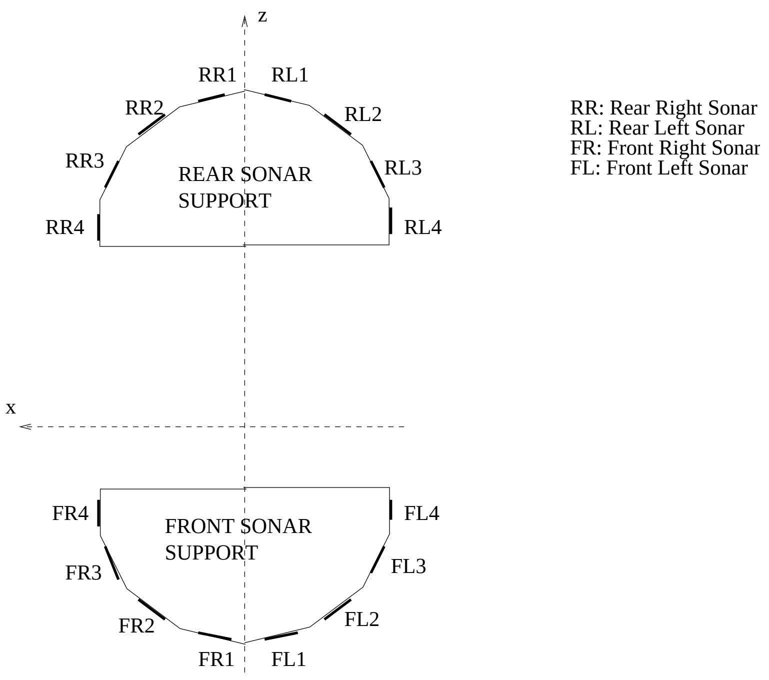


Figure 4.23: Sonar placement of the *Pioneer2* robot

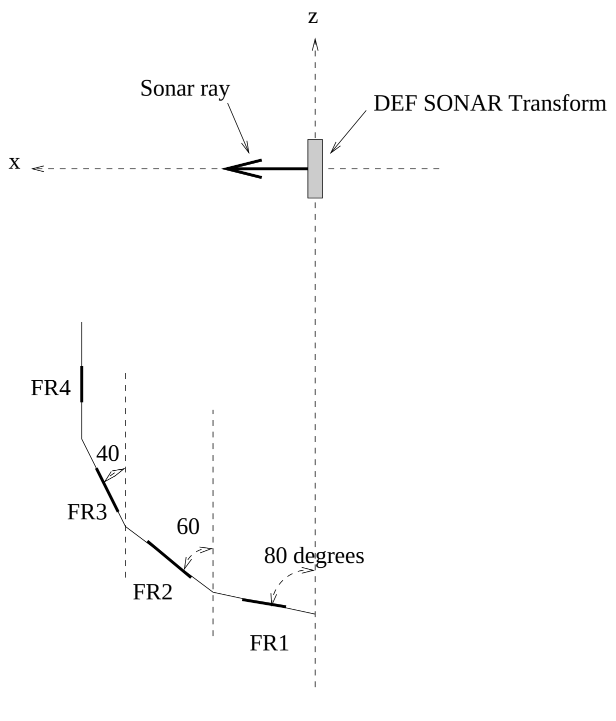


Figure 4.24: Angles between *Pioneer2* sonar sensors

To finish modelling the *Pioneer 2* robot, fill in the remaining fields of the `DifferentialWheels` node as hown in figure 4.25.

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 -0.116	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
FR4	0.136 0.209 0.160	0 1 0 0

Table 4.1: Translation and rotation of the *Pioneer 2* DEF SONAR Transforms

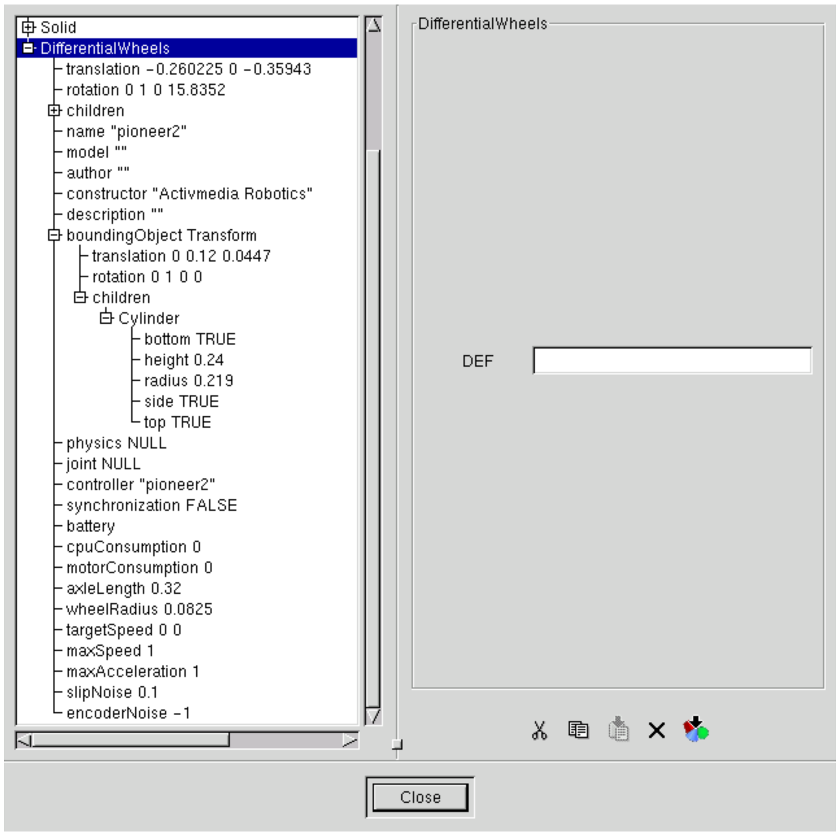

```
Solid
DifferentialWheels
  translation -0.260225 0 -0.35943
  rotation 0 1 0 15.8352
  children
  name "pioneer2"
  model ""
  author ""
  constructor "Activmedia Robotics"
  description ""
  boundingObject Transform
    translation 0 0.12 0.0447
    rotation 0 1 0 0
    children
      Cylinder
        bottom TRUE
        height 0.24
        radius 0.219
        side TRUE
        top TRUE
  physics NULL
  joint NULL
  controller "pioneer2"
  synchronization FALSE
  battery
  cpuConsumption 0
  motorConsumption 0
  axleLength 0.32
  wheelRadius 0.0825
  targetSpeed 0 0
  maxSpeed 1
  maxAcceleration 1
  slipNoise 0.1
  encoderNoise -1
```

Figure 4.25: Some fields of the *Pioneer 2* `DifferentialWheels` node

4.3.3 Controller

The controller of the *Pioneer 2* robot is fairly complex, using 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* 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".

4.4 Working with the robots from K-Team: Khepera, Koala, Alice

Three research robots: *Khepera*, *Koala* and *Alice* are produced by K-Team SA, as described on their Web site: http://www.k-team.com. The CD-Rom includes pre-defined models for these three robots. To use them, open the `.wbt` files in a text editor and copy/paste the whole `DifferentialWheels` node in your current `.wbt` file. A model for the *Khepera* robot can be found in the `khepera.wbt` file, a model for the *Koala* robot can be found in the `koala.wbt` file and a model for the *Alice* robot can be found in the `alice.wbt` file.

4.4.1 khepera.wbt

In Webots version 2.0, it was possible to model a gripper on a *Khepera* robot. This has been temporarly removed from Webots, but we intend to implement the gripper in a forthcoming version of Webots 3. Similarly, the transfer of Webots data directly to the real robot is no longer possible for the time being.

This model features a simple robot with the infra-red sensors implemented as `DistanceSensor` nodes.

It is possible to add a camera on top of the *Khepera* (like a K213 or K6300) by using a `Camera` node.

Additionally a communication system can be implemented by using a `Receiver` and / or an `Emitter` node.

4.4.2 koala.wbt

This file contains a simple model of the *Koala* robot with `DistanceSensor` nodes as infra-red sensors. As with the *Khepera*, it is possible to add new sensors and actuators.

4.4.3 alice.wbt

This file contains a simple model of an *Alice* robot, which can be extended as for the *Khepera* and *Koala* with, of course, much smaller devices...

Chapter 5

Robot and Supervisor Controller

5.1 Overview

A robot controller is a program usually written in C or Java used to control one robot. A supervisor controller is a program usually written in 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 home 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 `controllers` directory as the default controllers directory in the Webots preferences. Webots will first search for a controller in your default directory, and if not found will then look in its own controller directory. Now, in your newly created `controllers` directory, you must create a controller directory, let's call it `simple`. Inside simple, several files must be created.

- ◦ a number of C source files, like `main.c` which will contain your code.
- ◦ if you work on Linux you need a `Makefile` which can be copied (or inspired) from the Webots controllers directories.
- ◦ if you work on Windows, you can either create a `Makefile.mingw` to use GNU make, and then type `make -f Makefile.mingw` in a DOS console in the directory of the controller, or you can create a project file from the Dev-C++ software.

Note that in order to use `make` in Windows, you need to install the mingw environment (or Dev-C++, which includes mingw) and make sure that the `bin` directory of mingw is in your

`PATH` variable. To ensure this, you can edit the `C:\autoexec.bat` file and add a line like: `path=PATH;C̈:\Program Files\Dev-C++\bin¨`

On Linux, you can compile your program by typing make in the directory of your controller.

On Windows, you can either type "`make -f Makefile.mingw`" or use Dev-C++ to perform the compilation.

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 robots (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 robot controllers. Each request is divided into two parts: a 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 to 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

Before reading any sensor information, the sensor must be:

- 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.
- displayed: this is only useful for camera devices, to ensure that the image rendering is performed appropriately. It consists of associating a graphical object with the sensor. This graphical object will be updated automatically when the sensor is run. It must be done once, before entering the endless loop.
- 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.
- run: this is performed by the `robot_step` function inside the endless loop.
- 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, nor associated with graphical objects. To control an actuator, it must be:

- 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.

- set: this is performed by the appropriate `set` function specific to each actuator (see `differential_wheels_set_speed` for example). It is usually called in the endless loop with different computed values at each step.
- 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`. 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!