1 Lab+Hwk 1: Introduction to the Webots Robotic Simulator

This laboratory requires the following equipment:

- C programming tools (gcc, make).
- Webots simulation software.
- Webots User Guide
- Webots Reference Manual.

The laboratory duration is about 2 hours. Homework will be due on **the 6th day after your lab session, at 5 p.m**. Please copy your solutions, plots, descriptions (*.doc, *.ps, *.html, *.pdf, *.txt), movies, and so on into the directory *~correll/lab01/yourusername/*. The directory *~correll/lab01* allows you to write but not to read. The corresponding Webots files (see section 2 for details) should be saved in a sub- directory *webots/*. Please strictly follow these guidelines to facilitate the grading process, and keep in mind that no late solutions will be accepted unless motivated by health reasons (doctor certificate required).

1.1 Grading

In the following text you will find several exercises and questions. The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing a piece of code and performing a simulation. The notation $\mathbf{S_x}$ means that the question can be solved using only additional simulation. The notation $\mathbf{Q_x}$ means that the question can be answered theoretically without any simulation. **Please use this notation in your answer file**!

The number of points for each exercise is given between parenthesis. The total sum of the points is for the laboratory and homework parts is 100.

1.2 Khepera

Khepera is a miniature mobile robot developed to perform “desktop” experiments. It was developed at LAMI (EPFL) and is commercially distributed by K-Team SA, a spin-off company from the EPFL. Its distinguishing characteristic is its small size (5.5 cm in diameter). Other basic features are: significant processing power (Motorola® 68331, 32 bit processor running at 16 MHz), an EEPROM of 128 kB (which can be extended up to 512 kB) and a SRAM of 256 kB, energetic autonomy of about 45 minutes (with no additional turrets), precise odometry, and a belt of eight light and proximity sensors. The wheels are controlled by two DC motors with incremental encoders, and can rotate in both directions. Figure 1-1 shows a close-up of a Khepera robot.

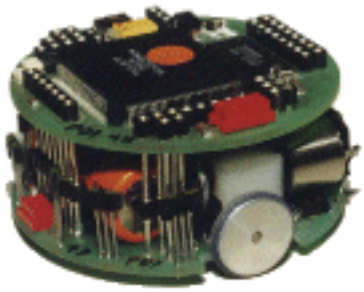

Figure 1-1: Close-up of a Khepera robot.

The simple geometrical shape and the positioning of the motors allow the Khepera to negotiate any kind of obstacle or corner. *Figure 1-2* shows the positions of the sensors and motors on the Khepera robots.

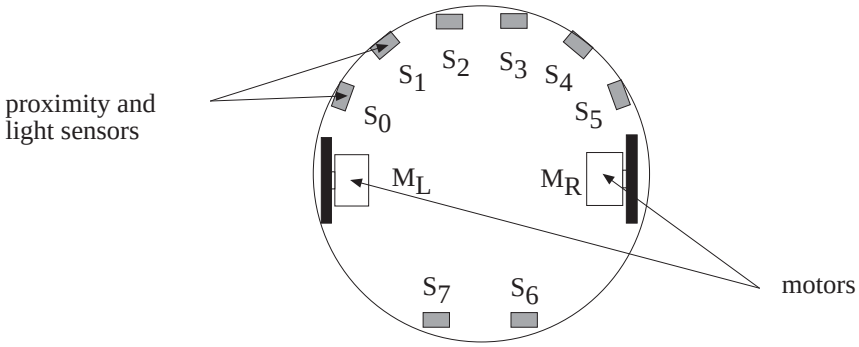

Figure 1-2: Position and orientation of sensors and actuators on the Khepera basic module.

Modularity is another characteristic of the Khepera robot. Each robot can be extended by adding a variety of modules. For instance, a possible extension module is the gripper which can grasp and carry objects up to 5.0 cm across and weighting up to 20 g. It has one degree of freedom (d.o.f.) in its arm elevation and one “binary” d.o.f. in its gripper (open or closed). The Khepera can be simulated in its basic form, and also with the extension modules, in the Webots simulator, as shown in *Figure 1-3*.

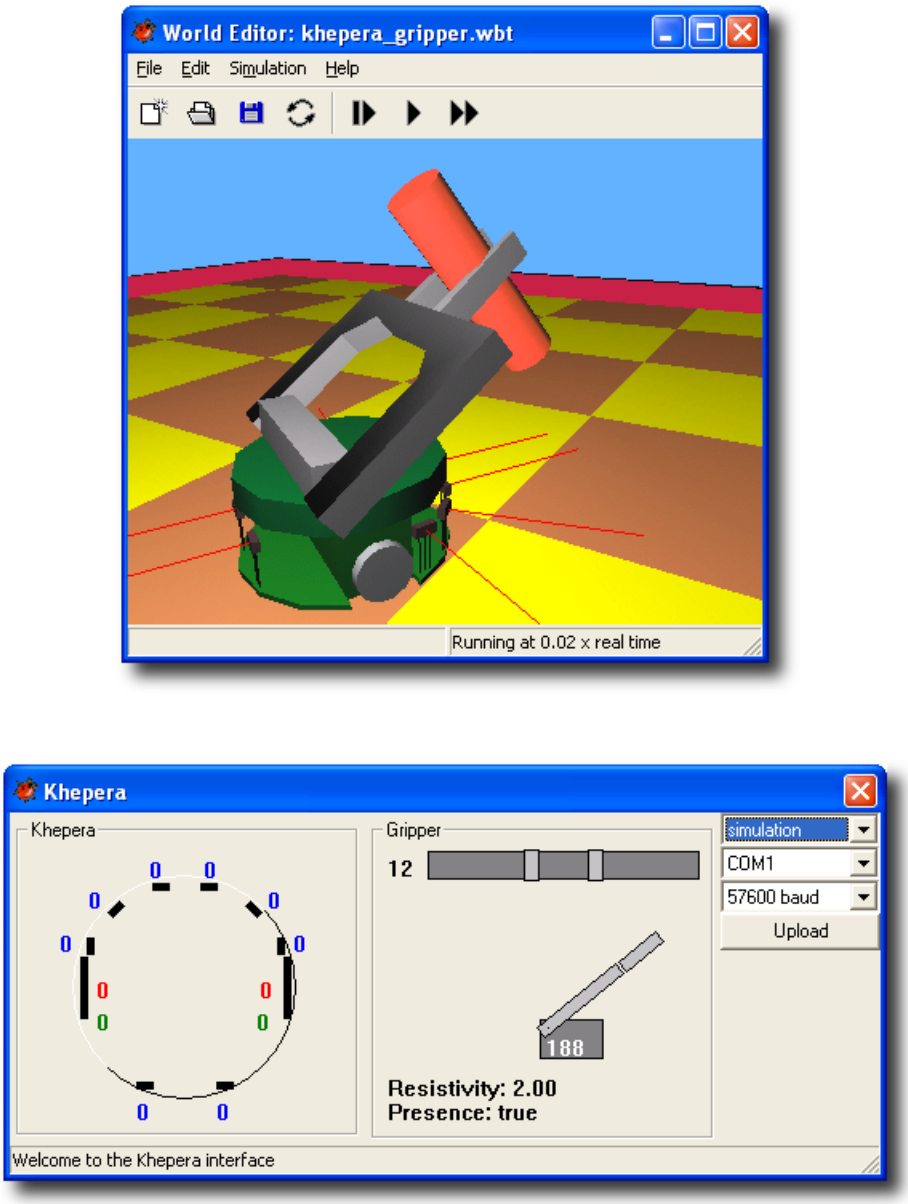

Figure 1-3: Webots simulation of the Khepera robot equipped with gripper turret.

1.3 Webots

Webots is a fast prototyping and simulation software developed by Cyberbotics Ltd., a spin-off company from the EPFL (LAMI lab).

Webots allows the experimenter to design, program and simulate virtual robots which act and sense in a 3D world. Then, the robots' controllers can be transferred to real robots (see *Figure 1-4*). Webots can either simulate the physics of the world and robots (nonlinear friction, slipping, mass distribution, etc.) or simply the kinematic laws. The choice of the level of simulation is a trade-off between simulation speed and simulation realism. However, in any case all sensors and actuators are affected by

a realistic amount of noise so that the transfer from simulation to the use of real robot controllers is usually very smooth.

Webots can simulate any types of robots, including wheeled, legged, flying and swimming robots. Several examples are included in the Webots distribution. During this laboratory we will perform experiments only with the Khepera models. Khepera robots can be equipped with several different turrets (Webots samples of the Khepera robot include different vision cameras and grippers). The environment editor allows one to design and place several types of objects in the environment.

A very interesting feature of the Webots simulator, in particular for multiple-robot experiments, is the way supervisor modules are implemented. A supervisor module is a program similar to a robot controller module. However, such a program is not associated with a given robot but rather with the whole simulation world. The supervisor module is able to get information from robots about their internal (control flag, sensor data, ...) and external variables (position, orientation, ...), and also from some objects in the world such as balls. Supervisor modules can be used for:

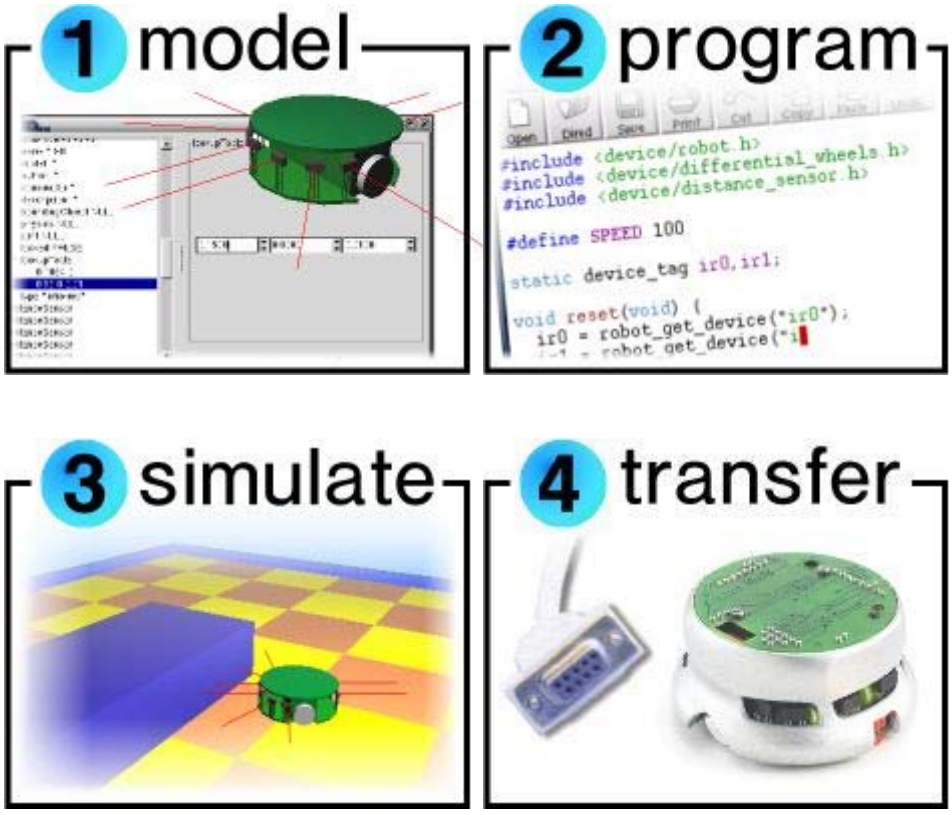

Figure 1-4: Overview of Webots principles.

- Inter-robot communications;
- Monitoring of experimental data (e.g., robot trajectories or sensor data);
- Controlling experimental parameters;
- Defining new virtual sensors.

Controller and supervisor modules are programmed in C, C++ or Java.

2 Lab: Getting familiar with Webots programming

This lab consists of several modules. We will start by programming a simple obstacle avoidance behavior for a single robot. We will then move to more complex collective behaviors that can help you to get insights for solving the homework problems.

Remark: In order to facilitate the grading of exercise set, copy all your **controller**, **supervisor**, and **world** modules in *~correll/lab01/youruser-name/webots/worlds* and *~correll/lab01/yourusername/webots/controllers* after you finish the exercise. World files contain all the information not only for recovering the environment but also the corresponding controllers and supervisor programs: **make sure that you save them and deliver as part of your solution even if not explicitly mentioned**! Answer to the theoretical questions, a description of where the files can be found and their names, plots, movies[1], and so on should be saved instead in the directory *~correll/lab01/yourusername*. Do **not** save your controller, supervisor, and world modules in this directory but use the appropriate subdirectories!

2.1 Getting started with Webots

Launch the simulator by entering the `webots &` command on your terminal. Create your own local webots directory as proposed by Webots. Configure the world and controller path with the *Edit->Preferences* menu as described in the Webots User Guide (chapter 3, section 3, subsection 2).

Using *khepera.wbt* as a template, create a maze in which you have a single Khepera robot without any extension turrets. Copy and paste boxes to add walls. Boxes can be resized from the scene tree window. If you have troubles, consult first the Webots User Guide, and then ask the teaching assistant. Save your environment as *myobstacle.wbt*. Hint: you can drag and drop objects by shift-clicking them.

Implement a program called *obstacle.cc* which allows the robot to avoid walls and move around the arena. You can start from the following example:

[1] Webots allows you to create mpeg movies: see chapter 3, section 3 of the User Guide for details. Usually, saving your world files is sufficient for our grading purposes. However, if you want to explain a particular effect you noticed or you want to save movies for your archives, we strongly encourage you to create such movies. Try to record brief sequences since the resulting size file can be huge. Space should not be a problem on your machine but ask your TA if you experience trouble.

obstacle.cc

```
#include <cstdio>
#include <device/robot.h>
#include <device/differential_wheels.h>
#include <device/distance_sensor.h>
#define NB_SENSORS 8

DeviceTag ds[NB_SENSORS];

static void reset(void) {
  char s[4]=”ds0”;
  for(int i=0;i<N_SENSORS;i++) {
    ds[i]=robot_get_device(s);
    s[2]++; /* increase the device number: “ds1”, “ds2”, etc. */
  }
}

void main() {
  int i,s;
  robot_live(reset); /* initialization */
  /* enable IR distance sensor reading every 64 ms */
  for(i=0;i<NB_SENSORS;i++) distance_sensor_enable(ds[i],64);
  for(;;) { /* endless loop */
    s = distance_sensor_get_value(ds[3]);
    printf(“proximity sensor value #3 = %d\n”,s);
    differential_wheels_set_speed(10,5);
    robot_step(64);
  }
}
```

Remark: Such a program, once compiled and linked, cannot be run from a shell like a standard C program in UNIX. It has to be launched within the simulator because there is communication with Webots through pipes.

Functions of type `xxx_enable` initialize and enable the Khepera sensors. Functions of type `xxx_disable` disable sensors. Functions of type `xxx_get_xxx` read sensors values while those of type `xxx_set_xxx` send commands to actuators. Finally, the function `robot_step` implements one simulation iteration, which in this case corresponds to 64 ms of real time.

You will find this simple program in the directory *~correll/webots/controllers/obstacle/*. In order to compile and link it, simply enter `make`. The command `make` executes a script contained in the dedicated *`Makefile`* of the directory.

You can now execute your controller in Webots. Open the scene tree window, then click on a robot in your Webots 3D view to select the robot in the scene tree, choose the *controller* field of the *`DifferentialWheel`* robot corresponding to your Khepera robot and load the controller named *obstacle*. Finally, you can simply push the *play* button in the Webots main window to execute your obstacle avoidance

program. Remember that with the *step* button you can execute your program step-by-step.

I_1 (10): As you may have noticed, the original version of *obstacle.cc* does not allow the robot to avoid obstacles very well. Modify the original program so that your robot is able to avoid any type of obstacle in any maze. Create a new directory called *myobstacle* in *webots/controllers*. Copy the contents of *obstacle* into this new directory, and save your changes in *myobstacle.cc*[2]. Update the robot controller and save your world again as *myobstacle.wbt*. Hint: implement an appropriate perception-to-action loop by starting with the 2 central sensors on the front and then exploiting all the 8 available proximity sensors. Try to keep your solution simple and reactive.

S_2 (5): Test the robustness of your obstacle avoidance algorithm: put your robot in the middle of a U-shaped wall which you will create and save as *Uobstacle.wbt*: what happens? What is important for avoiding this type of deadlock?

I_3 (5): Modify your program so that the robot is able to escape from a U-obstacle. Save your new program in *Uobstacle.c*. Save your world as *Uobstacle.wbt*.

S_4 (5): Test the robustness of your program again by painting all the walls of the U-obstacle maze (from **I_3**) black, and again with them painted white (hint: Black: R = G = B = 0, White: R = G = B = 1). What happens? How does the color affect the obstacle avoidance behavior of the robot? Save the world as *UobstacleB.wbt* and *UobstacleW.wbt*.

S_5 (5): Test the effectiveness of your avoidance controller by introducing 3 or 4 other robots into your original maze (from **I_1**). What happens? What are the additional difficulties involved in avoiding other robots (list at least two)? Save the world as *multiobstacle.wbt*.

Q_6 (5): Do you have an idea how a proximity sensor works and how could it be implemented from a hardware point of view?

2.2 Braitenberg vehicles

Valentino Braitenberg presented in his book “Vehicles: Experiments in Synthetic Psychology” (The MIT Press, 1984) several interesting ideas for developing simple, reactive control architectures and obtaining several different behaviors. These types of architectures are also called “proximal” because they tightly couple sensors to actuators at the lowest possible level (proximal to the hardware). Conversely, “distal” architectures imply the presence of at least one additional layer between sensors and actuators, a layer which could consist for instance of basic behaviors, based on a priori “knowledge” which the programmer wants to give the robot.

[2] In order to import controllers in webots, directory name and file source name should be equal (e.g., *filename* contains *filename.c*). We will implicitly assume this change from now on each time that we ask you to save a given solution with a given file name.

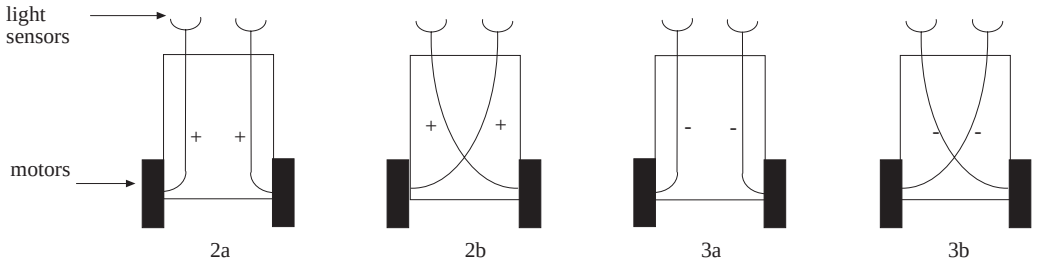

Figure 2-1: Four Braitenberg vehicles. The numbering corresponds to the original one used by Braitenberg. Plus signs correspond to excitatory connections, minus signs to inhibitory ones. Vehicle 2a avoids light by accelerating away from it. Vehicle 2b exhibits a light approaching and following behavior, accelerating more and more as approaches the source. Vehicle 3a avoids light by braking and accelerating away toward darker areas. Finally, vehicle 3b approaches light but brakes and stops in front of it.

In this lab, we want to see what we can achieve if robots are programmed as Braitenberg vehicles. *Figure 2-1* shows four different Braitenberg vehicles.

2.2.1 Phototaxis behaviors

Figure 2-2 shows vehicles 2a and 2b of *Figure 2-1* in front of a light source. Vehicle 2a can be considered "shy" because it moves away from the source, hiding in the darkness. Conversely, vehicle 2b can be considered "aggressive" because as soon it sees light, it approaches it until it hits the source.

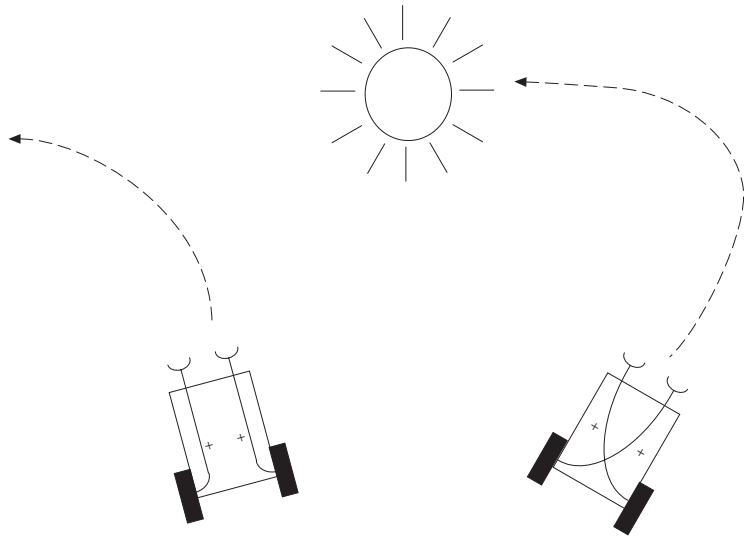

Figure 2-2: Behavior of two different Braitenberg vehicles with excitatory connections in front of a light source.

Remark: the original Braitenberg vehicles use omni-directional analog sensors. In the example of *Figure 2-2*, the different excitation of motors is computed on the difference between sensor stimuli according to their scalar distance from light source. On mobile robots such as Khepera, sensors can measure the light intensity (or the reflected light intensity, if

proximity sensor instead of light sensor are used) only in a given cone: they are directional.

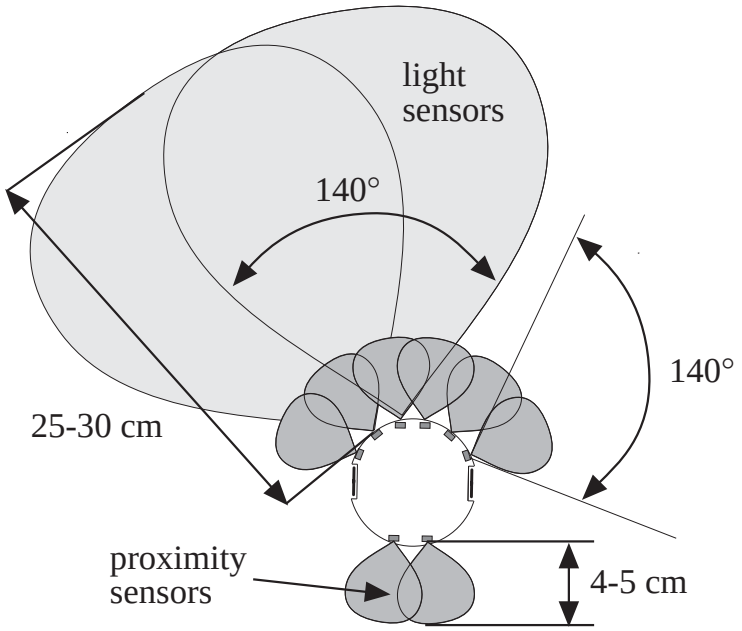

Figure 2-3: Directionality and range of Khepera light (a) and proximity sensors (b). In Webots lobes are approximated by rays.

In order to have the exact same experimental conditions as presented in Braitenberg's book, we should raise the light source (remember that lamps can be positioned at any height above the arena in the Webots simulator) so that is visible from any point and provide omnidirectional signals to the robot. However, Braitenberg vehicles work correctly, and sometimes even better (enhanced differences in the sensor stimuli), with directional sensors too.

I$_7$ (10): Program both behaviors shown in *Figure 2-2*. Hint: place the light source at a height of about 2-3 cm. To do that, create a new world with a `PointLight` node set in the middle of the arena. Program your own *braiten2a/* and *braiten2b/* controllers. Take inspiration from the example *obstacle.c* by substituting the `distance_sensor_xxx` functions with `light_sensor_xxx` functions and simply modify the coefficient values and/or signs of your sensors-to-actuators matrix. To do that, first copy the whole *myobstacle/* directory, rename it and include *light_sensor.h* into your source code. Save your file with the name *braiten2a.c* and *braiten2b.c*. Save your worlds as *braiten2a.wbt* and *braiten2b.wbt.*

3 Hwk: Understanding distance sensor modeling in Webots

In Webots, the behavior of distance sensor is modeled by computing the intersection between a ray cast from the center of the sensor and directed according to the orientation of the sensor (`rotation` field of the `DistanceSensor` node). For

each sensor a lookup table is used to determine the answer of the sensor according to the distance to the obstacle. This also defines implicitly the range of the sensor. This lookup table includes the return values as well as the noise level for a number of distances entries. A linear extrapolation is performed on this lookup table to compute the answer (value and noise) for a given measurement. More information about the modeling of distance sensors is provided in the user guide (kiki tutorial) and the reference manual (DistanceSensor node definition and API). Please be sure you understood the principle of such a distance sensor.

S_8 (10): As you may have noticed, a special case is made for infra-red distance sensor, since their behavior depends on the color of the objects. How does one mark a distance sensor to be an infra-red sensor? How does object colors influences the answer of such infra-red sensors?

S_9 (10): Look at the DistanceSensor nodes used to simulate the sensors of the Khepera robot. How many entries contains the lookup table? What is the maximum range of the sensors? Compare this lookup table to the figure describing the answer of the distance sensor given below (form the Khepera user manual). Does it fit?

Modeling a distance sensor with a single ray casting may be a bad approximation for sensors with a wide field of view, as the sensors of the Khepera robot. In most cases, such an approximation is sufficient and provides a fast simulation speed, but in some cases, it may be desirable to have a better sensor modeling. In order to improve this simple model, several options are available.

One may first use several simulated several distance sensors to approximate the behavior of a single real distance sensor. All the simulated distance sensors should be located at the same position, but heading into slightly different directions. This corresponds to several ray castings and the measured values for each sensor should be used to compute a weighted sum corresponding to a more realistic measurement for the real distance sensor.

I_{10} (15): Model infra-red sensors with 3 ray castings (3 DistanceSensor nodes per real sensor) on the Khepera robot. Describe the angles between the different directions of these simulated sensors. Which weights did you chose to compute the weighted sum? Why? How does this affect the behavior of the obstacle avoidance controllers?

Another possibility for creating a more accurate model for distance sensor is to use the `Camera` node in `range-finder` mode. In this mode, the `Camera` node will not return a standard camera image corresponding to the viewpoint of the camera, but instead it will return a depth image corresponding to the viewpoint of the camera. In such a depth image, each pixel doesn't contain any color information, but instead it contains depth information. Of course, modeling a camera is computional expensive and we probably do not need high resolution images to model distance sensors. Hence a low resolution image (like 4x4 pixels) should be sufficient to perform fairly rapidly 16 rays casting that should be averaged through as weighted sum as performed previously.

Read carefully the documentation of the `Camera` node in the reference manual (node description and API) to understand how to set up and use a `Camera` node in

`range-finder` mode. You may want to set the `display` field of the `Camera` node to `FALSE` to avoid having a camera window popping up.

I_{11} (15) Replace the `DistanceSensor` based model used previously by a `Camera` based model. Explain your design choices (weight parameters, field of view of the camera, etc.). How does this new modeling affect the behavior of the robot comparing to the 3 distance sensors model and to the single distance sensor model. Compare the simulation speed of all three solutions.

S_{12} (5) According to you, what is the best option for modeling distance sensors for this obstacle avoidance task? Why?