MAURICIO ZEPEDA

BACHELOR OF SCIENCE THESIS

BACHELOR OF SCIENCE THESIS

MAURICIO ZEPEDA

The Rat's Life Environment for Research and Education

Unmanned Vehicle Centre
Department of Mechanical Engineering
Royal Military Academy

Intelligent Robotics Laboratory
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics

January 2012

Mauricio Zepeda: *BACHELOR OF SCIENCE THESIS,* The Rat's Life Environment for Research and Education, BSc. in Software Engineering,

SUPERVISOR:
Dr Ir Eric Colon

LOCATION:
Brussels, Belgium

TIME FRAME:
January 2012

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth

Dedicated to my friends and family, who have always provided me
with unconditional support.

ABSTRACT

In the following pages I will explore the field of mobile robotics. I will focus on the algorithmic side of mobile robots, in particular on the basic algorithms used within the examples provided on the robot simulation software *Webots*, and on the cognitive benchmark titled *Rat's Life Programming Contest*. This contest is included within the simulation software as a way to measure the quality of an assembly of algorithms, and also to motivate beginning students into the advanced field of mobile robot algorithm design.

RESUME

Dans les pages suivantes je vais explorer le domaine de la robotique mobile. Je vais me concentrer sur le coté algorithmique des robots mobiles, en particulier sur les algorithmes de base utilisés dans les examples fournis par le logiciel de simulation de robots *Webots*, et sur l'épreuve intitulée *Rat's Life Programming Contest* (Concours de la programmation de vie des rats). Ce concours est inclus dans le logiciel de simulation comme un moyen de mesurer la qualité d'un assemblage d'algorithmes, et aussi pour motiver les étudiants débutants dans le domaine avancé de la conception algorithmique du robot demobile.

ACKNOWLEDGMENTS

Many thanks to all the people who have helped me along the way.

Regarding my studies at the Florida Institute of Technology (*FIT*), many thanks go to Dr. William D. Shoaff, Dr. Ryan Stansifer, Dr.Phillipe K. Chan, Dr. Ronaldo Menezes, Dr. Keith Gallagher, and the other professors who have also touched my life.

I would also like to thank my friend Erdal Tuleu who provided me with valuable guidance during my first programming steps.

Regarding the Sensing Technology and Robotic Systems (*STARS*) study abroad program, I would like to thank Dr. Charles Bostater, Dr.Bálint Kiss, Mrs. Zsuzsanna Bagaméri, Dr. Csákány Rita, Dr.Adamis Gusztáv, Dr. Attila Sali, Dr. Nathan Lemons, Dr. Eric Colon, Dr. Xavier Neyt, and Dr. Wim Mess for their patience, teachings, and direction.

I would also like to thank all the people involved in the administration of this program because without them none of this would have been possible.

CONTENTS

LIST OF FIGURES

LISTINGS

Part I

BACKGROUND

1 INTRODUCTION

1.1 DEFINITIONS

There are many different ways to say what a robot is. For the purpose of this work I will use the definition of *robot* from the Cambridge online dictionary.

ROBOT: A machine used to perform jobs automatically, which is controlled by a computer.

MOBILE ROBOT: A robot able to transport itself in a given environment.

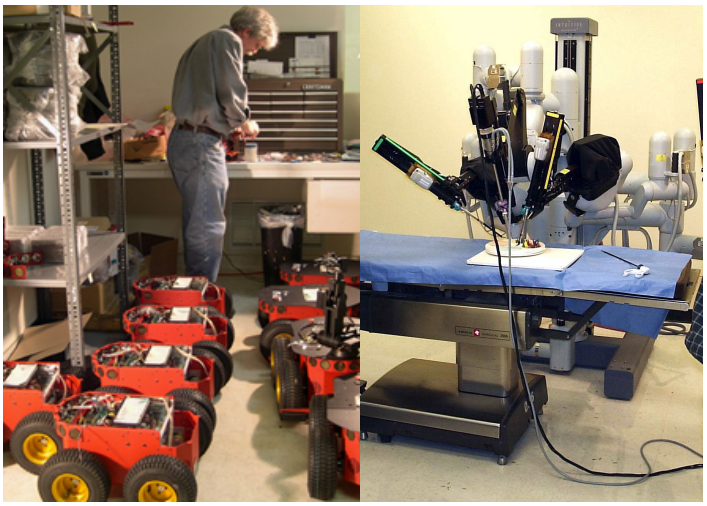
1.2 ROBOTS AND COMPUTERS

The beginnings of (modern) robotics are closely tied to those of computers, for they are necessary to their existence. In this line, the model of Alan Turing in 1936 of a programmable machine must have probably given rise to the notion of a self-programming machine or that of a machine programmed to make optimal decisions (i.e. artificial intelligence). These machines would allow for the development of algorithms that could solve complex tasks which require autonomous robots to be "intelligent". All in all, without programmable computers it would not be possible to even imagine the construction of robots as we know of them today.

1.3 MOBILE ROBOTS

As opposed to fixed robots, mobile robots are able to move their position within their environment. Not only wheeled robots, humanoids, and unmanned air vehicles are mobile robots. But also lesser known examples like autonomous underwater vehicles, or polar robots—which are able to navigate on top of ice. The field of fixed (i.e. non-moving) robots is mainly composed of jointed arm robots. This type of robots are frequently used in automotive, manufacturing, medical, and spacecraft settings. Although not able to move, fixed robots can achieve the level of precision required to perform medical surgery.

Mobile robots are a very prominent area of research in robotics due to the value of their promising applications. Specially in the military, industry, and security fields. However, today it is also possible to acquire mobile robots from the consumer market.



(a) Mobile Robots (b) Fixed Robots

Figure 1: A simple classification of robots

1.4 MOBILE NAVIGATION

Mobile robots can be further classified by their navigation strategy:

NAVIGATION STRATEGY	DESCRIPTION
tele-op	is operated remotely
guarded tele-op	is operated remotely and avoids collisions
line-following	moves so that a visual line stays centered on a sensor
autonomously randomized	navigates randomly and avoids collisions
autonomously guided	localizes itself to reach checkpoints
sliding autonomy	provides two or more of the above

Table 1: Different types of mobile robots

Usually a *sliding autonomy* is composed of at least an *autonomously guided* mode, and a *guarded tele-op* mode to avoid human harm. Programming a guarded tele-op is relatively easy since all the navigation commands are done remotely. The biggest concern is getting the translation of commands to the right mechanical movements, as well as the problems related to the transmission of the sensor's data to the remote controller. Both of this problems vary from robot to robot, but they do not depend on the context. Accordingly, it is possible to encapsulate this problems behind a high-level interface.

Ideally, an *autonomously guided* robot should also be programmed to be context-independent. However, due to the complexity of the problems involved; this has not yet been achieved. More specifically, the problem of "what to do when the sensors see something" is not trivial. And it depends directly on what "something" means.

To know what something means requires understanding; and while "understanding" is the comprehension of knowledge—knowledge is subjective to the person that defines it. Even if we agree on one single body of knowledge, such a science, and the robot is able to "understand" what an object is—for example; the question of "what to do with it" still remains allegedly subjective. It is for this reason that there cannot exist a universally intelligent computer.

"One man's trash is another man's treasure" is a known idiom that depicts the subjectivity of purpose.

Nevertheless, robots have been getting surprisingly good at well defined tasks. For example the tech giant Google is working on an autonomously guided car which has already proved to be able to drive itself through fast-moving traffic in a highway [1].

The lead engineer behind this project, Sebastian Thrun, also won the Grand Challenge of the Defense Advanced Research Projects Agency in 2005. He and his Stanford University team built the fastest cars able to drive autonomously on a path of over 132 miles of desert terrain. [10]

Figure 2: "Stanley" won the DARPA Grand Challenge in 2005

1.5 ALGORITHMS

Aside from the complex control mechanisms that Stanley required to win the challenge; it also shared some of the problems common to all mobile robots, and to anyone who desires to navigate. The autonomous navigation problem consists of the questions:

- Where am I?
- Where am I going?

When a mobile robot does not have any sort of external help the answers to this questions can get increasingly complex. The solution to this problems involves algorithms such as visual recognition, intelligent classification, learning, self-localization, map building, and path planning. Moreover, all of this algorithms need to be assembled in a special way, and this can be further considered an algorithm in itself.

1.5.1 *Limitations*

The performance of some algorithms experiences an exponential growth with relation to the size of the input of the problem. In some cases a small problem may require astronomical amounts of computing power to be perfectly solved (in an exhaustive way). An example of this is a chess-playing robot that is able to visualize all possible game states.
Although this kind of problems are not yet known to be solvable, different optimal guessing techniques (i.e heuristics) are used so that the algorithm considers only those possibilities that matter. This is often the case with algorithms for mobile robots and it is also the reason why finding the right heuristics is one of the main concerns in mobile robot algorithm design. On the other side this is also the reason why the search for efficient algorithms that do not rely on heuristics is of high priority in the Artificial Intelligence field.

1.6 GOALS

I will describe some of the most basic algorithms used to solve the autonomous navigation problem. The *Webots* mobile robot simulation toolkit includes a benchmarking environment which is specially designed to measure and explore the autonomous navigation problem. This environment has been physically reproduced with the use of Lego© blocks and affordable e-puck robots at the École Polytechnique Fédérale de Lausanne (*EFPL*) in Switzerland. Although the official website of the contest contains the physical building instructions, the site is no longer maintained and the instructions are not guaranteed to work.

The simulation environment includes a basic assembly of algorithms as a starting point for the contest. I will explain how they work, and I will attempt to improve them with basic modifications.
This should serve as a starting point for interested students on autonomous robot navigation algorithm design.

2 TOOLS

2.1 THE WEBOTS MOBILE ROBOT SIMULATOR

The Webots software is an Integrated Development Environment (*IDE*) specifically tailored for fast mobile robot prototyping with realistic physical simulations. It has been developed by *Cyberbotics Ltd*. sine 1996 [3]

The 3-dimensional model of the environment is defined in a world file (`.wbt`) that contains information about each of the objects' position, orientation, geometry, appearance, and physical properties. It also specifies the name of the controller in charge of each robot.
It is possible to model the world in other computer-aided design (*CAD*) software applications which are able to export into a VRML97 format and then use the VRML97 file importer to translate the file into the `.wbt` world format—which looks familiar to the VRML format. The user interface also includes a hierarchical *tree editor* to satisfy the basic 3-d modeling requirements.

The robot controllers can be programmed in the C, C++, Java, Python, Matlab, or URBI programming languages. Depending on the robot model, it may be possible to remote control it by running the program in the local PC which takes the sensors' data as input and sends commands directly to the robots' actuators. Additionally it is possible to cross-compile the program to the micro-controller's low-level language, and thus enable a strictly autonomous scenario.

The simulation uses the open source "Open Dynamics Engine" (*ODE*) to perform the physical dynamics simulation. The main benefits of using this project is that it is mature and platform independent. This also means that a certain degree of reliability can be safely expected from the simulations.

2.1.1 *Advantages*

The software includes predefined models for commercially available robots including[6]:

- Aibos
- Bioloids
- Boe-Bot

- e-puck
- HOAP-2
- iRobot
- Create
- Katana
- Kondo KHRs
- Nao
- Pioneer
- Shrimp III
- Surveyor SVR-1

In addition to this models, the software also provides a large collection of sensors including[6]:

- Distance sensor
- Light sensor
- Cameras
- LIDARs
- GPS
- Gyrometer
- Accelerometer
- Compass
- Bumpers
- Force-sensor

Considering all this tools, it is not only faster and simpler to focus on the task at hand; but it is also possible to isolate the task of designing robotic algorithms without concerning about the physical limitations of the robot and the environment.

Because of the significant savings in both time and money that the toolkit provides, the *Webots* mobile robot simulation environment has already been adopted by more than 750 universities and research centers worldwide [7].

2.1.2 *Documentation*

Cyberbotics Ltd. actively maintains three sources of documentation[4]:

- Reference manual: provides a detailed and hierarchical description of every functionality that the software exposes.
- User guide: provides a step-by-step description of the software capabilities, as well as a summary for each of the sample projects.
- Robot curriculum: provides a learning path in mobile robotics with the use of *Webots* and the *e-puck* for the final goal of preparing a student to participate in the Rat's Life Programming Contest.

2.1.3 *Sample Projects*

In the following section I will show screen captures of some of the sample projects included with the software in order to illustrate its potential:

2.1.3.1 *Robotis DARwin-OP*

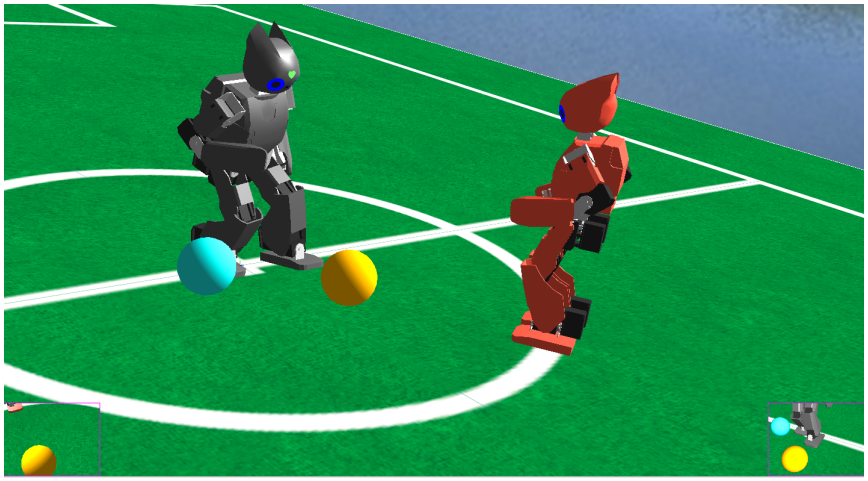


Figure 3: A ball-kicking robot

The humanoid robots use the DARwin-OP higher-level walk control library to kick a ball.

2.1.3.2 *iRobot Create*

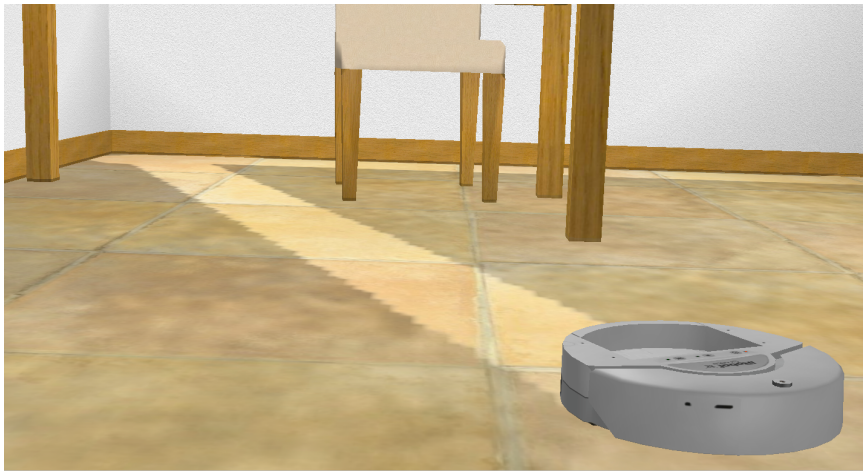


Figure 4: A vacumming robot

While the autonomously randomized robot navigates around the dirty floor a *supervisor* updates the environment to remark the clean path. The use of a *supervisor* is restricted to the *PRO* license.

2.1.3.3 *Sojourner Rover*

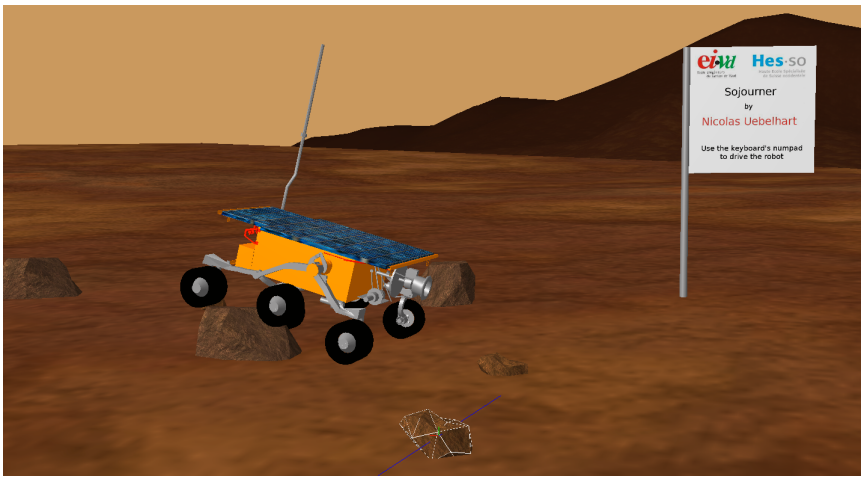


Figure 5: A planet-exploring robot

A wheeled robot navigates on Mars-like terrain. It is possible to control it with the keyboard.

2.1.3.4 *Salamandra Robotica*

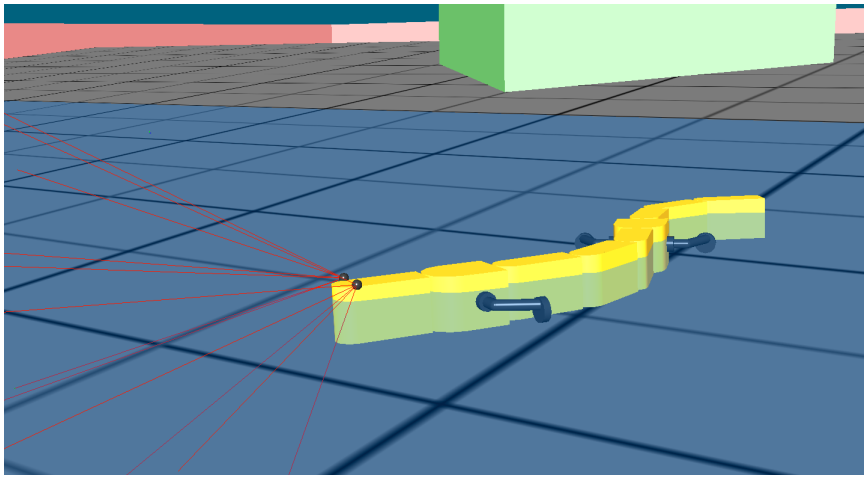


Figure 6: A swimming robot

A physics plugin is used to simulate hydrostatic and hydrodynamic forces. The robot uses two types of locomotion: walking on the ground; and swimming on the water.

2.1.3.5 *Autonomous Vehicle*

Figure 7: A street-driving robot

An autonomously guided robot brakes as it approaches a curve. It is possible to monitor the input of the corresponding sensors—in this case the front camera and the speedometer.

2.1.3.6 *Fujitsu HOAP-2 sumo*

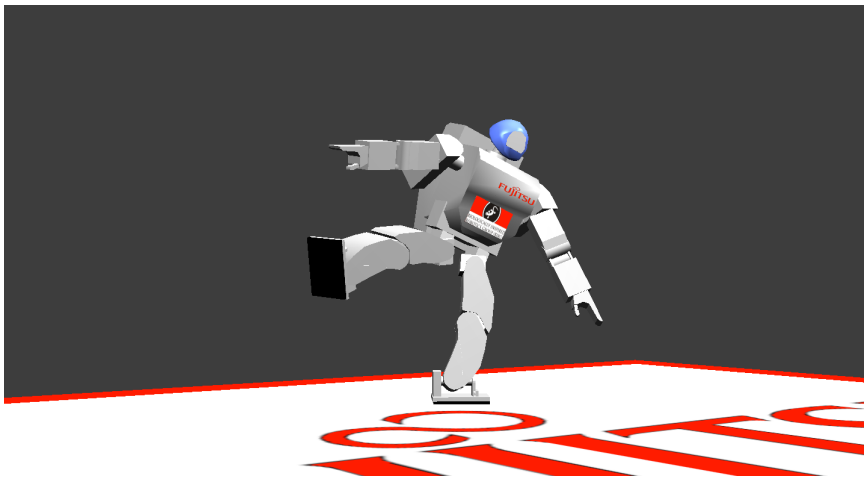


Figure 8: A sumo-dancing robot

A humanoid robot is able to maintain its balance while performing a Shiko dance. Touch sensors measure the pressure between the feet soles and the ground.

2.1.4 *User Interface*

Even though the files can be modified with third-party editors, and the console output may be redirected to another terminal; the *IDE* provides all the necessary tools to effectively develop a simulation.

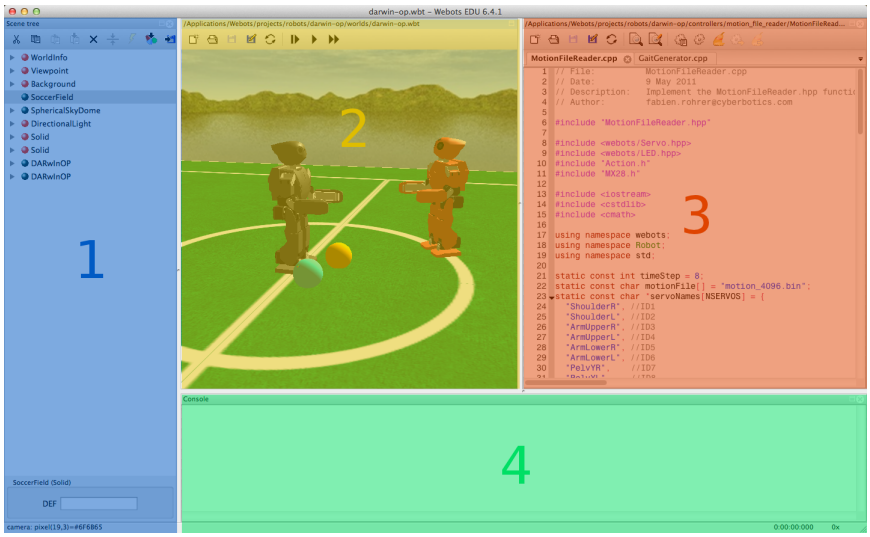


Figure 9: User Interface Layout

The default layout is separated into four parts:

1. Scene tree
2. Simulation
3. Text editor
4. Console

(contd...)

2.1.4.1 *Scene Tree*

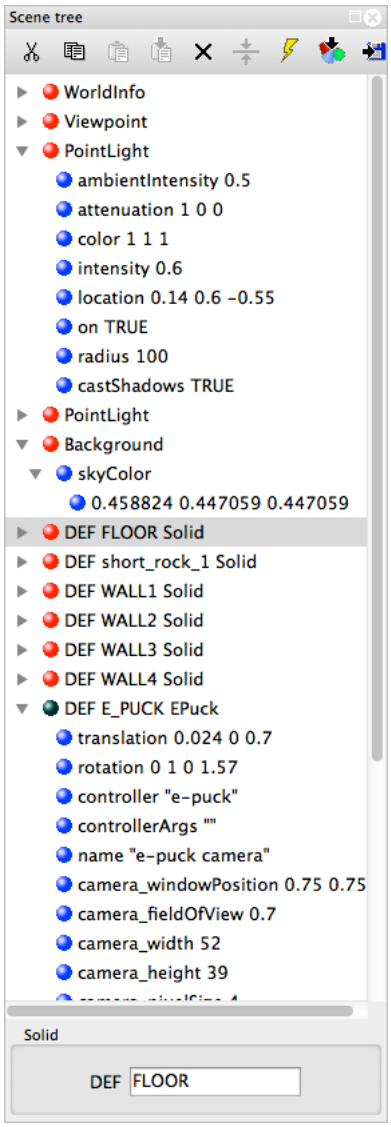


Figure 10: Scene tree window

A VRML-like structure representation of a `.wbt` world file is manipulated by the *scene tree* editor. The tree is composed by nodes which can contain field-value pairs, other nodes, or nothing.
Double-clicking on a field allows to modify its value, and double-clicking on a node toggles between the hide and show states.

2.1.4.2 *Simulation*

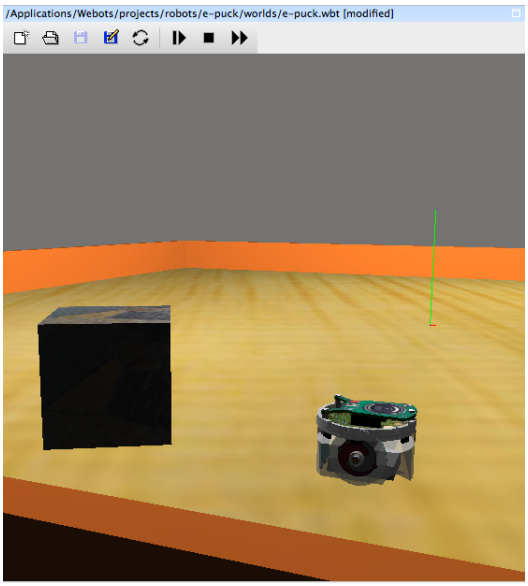


Figure 11: Simulation window

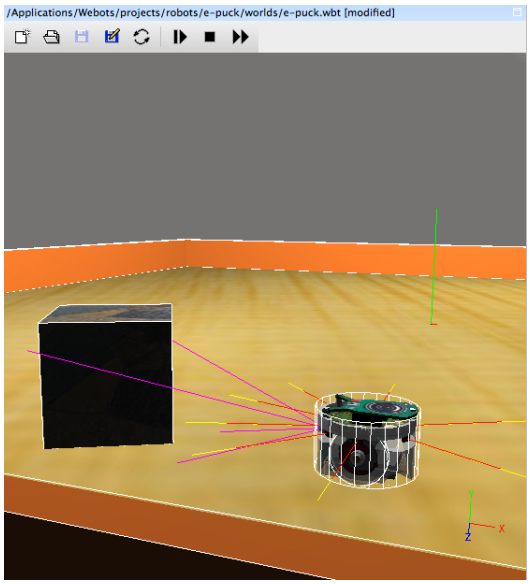


Figure 12: Simulation window with additional view options enabled

The visual representation of the 3-dimensional space is shown through a camera. The position, rotation, and tilt of the camera can be intuitively adjusted with the mouse buttons and wheel. Some of the buttons on the toolbar allow the user to:

- Advance one step
- Continuously loop through the simulation
- Stop the simulation

- Go through the simulation as fast as possible (requires *PRO* license)
- Restart the simulation

Other additional options reachable through the view menu allow the user to highlight the sensors' reach along with other useful metadata.

2.1.4.3 *Text Editor*

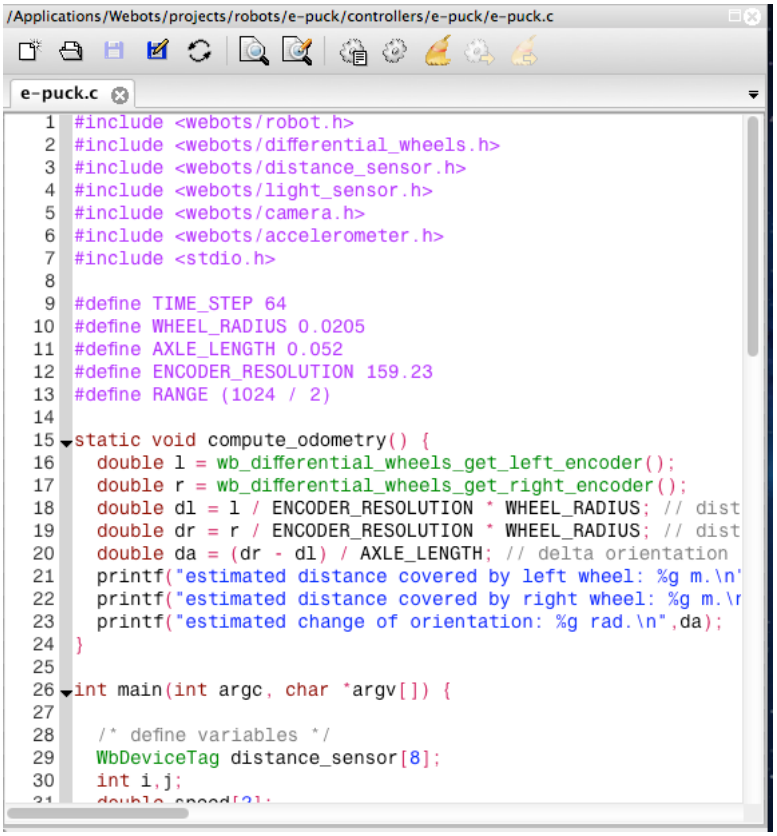


Figure 13: Code editor

The built-in text editor provides with convenient line-numbering and syntax highlighting. It supports multiple tabs, and block hiding. Some of the buttons on the toolbar allow the user to:

- Revert back to the original file
- Find and replace basic regular expressions
- Compile
- Cross-compile

2.1.4.4 *Console*

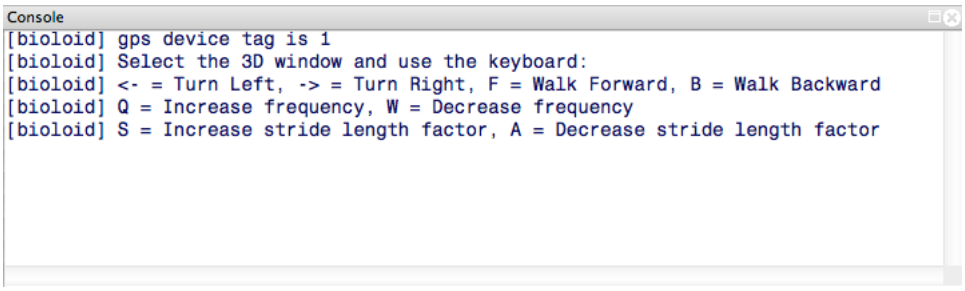


Figure 14: Console

The console displays the standard output and error streams of the program. It is possible to instead display them in the shell where the *Webots* application was started from by setting the `WEBOTS_STDOUT` and `WEBOTS_STDERR` environment variables to `1` [4].

2.1.5 *Development Workflow*

The integrated environment provides four convenient development stages:

1. Model: Define the 3-dimensional space using a hierarchical node editor.
2. Program: Use the built-in code editor to program a controller for an object.
3. Simulate: Test the controller in the virtual simulation.
4. Transfer: Move the controller program to the real robot.

This setup allows the programmer to find errors early and iterate fast. Ideally the development workflow then becomes: model, program, simulate, program, . . . , simulate, transfer. In order to use this convenient workflow there needs to exist a precise mapping from the simulation to the real robot. Cyberbotics [3] claims that the included model of an e-puck robot has been accurately calibrated. This may be one of the reasons why the e-puck was chosen for the Rat's Life Programming Contest. This workflow allows for even faster prototyping by ignoring the modeling stage, and guaranteeing the expected behavior on the real robot.

2.2 THE E-PUCK ROBOT

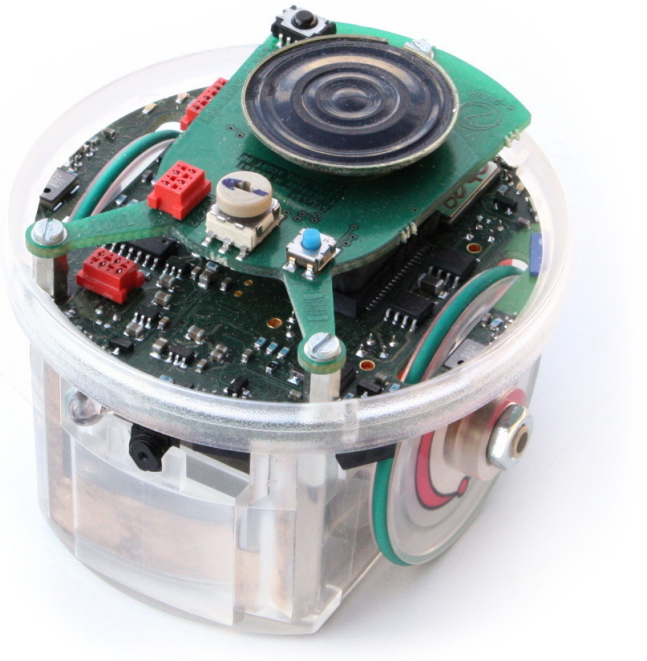


Figure 15: The e-puck robot

The e-puck robot was designed by Dr. Francesco Mondada and Michael Bonani in 2006 at the *EFPL*. All of its hardware and software components are fully open source as a way to encourage the development of the project's community. [5]

The robot was designed to be[5]:

- Simple
- Multi-purpose
- Extensible
- Robust
- Maintainable
- Simulation friendly
- User friendly
- Affordable

2.2.0.1 *Specifications [5]*

FEATURE	TECHNICAL INFORMATION
Size	around 7 cm of diameter
Battery	around 3 hours with the provided 5Wh LiION rechargeable battery
Processor	Microchip dsPIC 30F6014A @ 60MHz (about 15 MIPS)
Motors	2 stepper motors with a 20 steps per revolution and a 50:1 reduction gear
IR sensors	8 infra-red sensors measuring ambient light and proximity of obstacles in a range of 4 cm
Camera	color camera with a maximal resolution of 640x480 (typical use: 52x39 or 640x1)
Microphones	3 omni-directional microphones for sound localization
Accelerometer	3-dimensional accelerometer
LEDs	8 red LEDs on the ring and one green LED in the body
Speaker	on-board speaker capable of playing WAV or tone sounds.
Switch	16 position rotating switch
Bluetooth	Wireless communication
Remote Control	infra-red LED for receiving standard remote control commands
Expansion bus	expansion bus to add new possibilities to your robot.
Programming	C programming with the GNU GCC compiler system
Simulation	use Webots for simulation, remote control and cross-compilation.

2.2.1 *Electronic Circuit [5]*

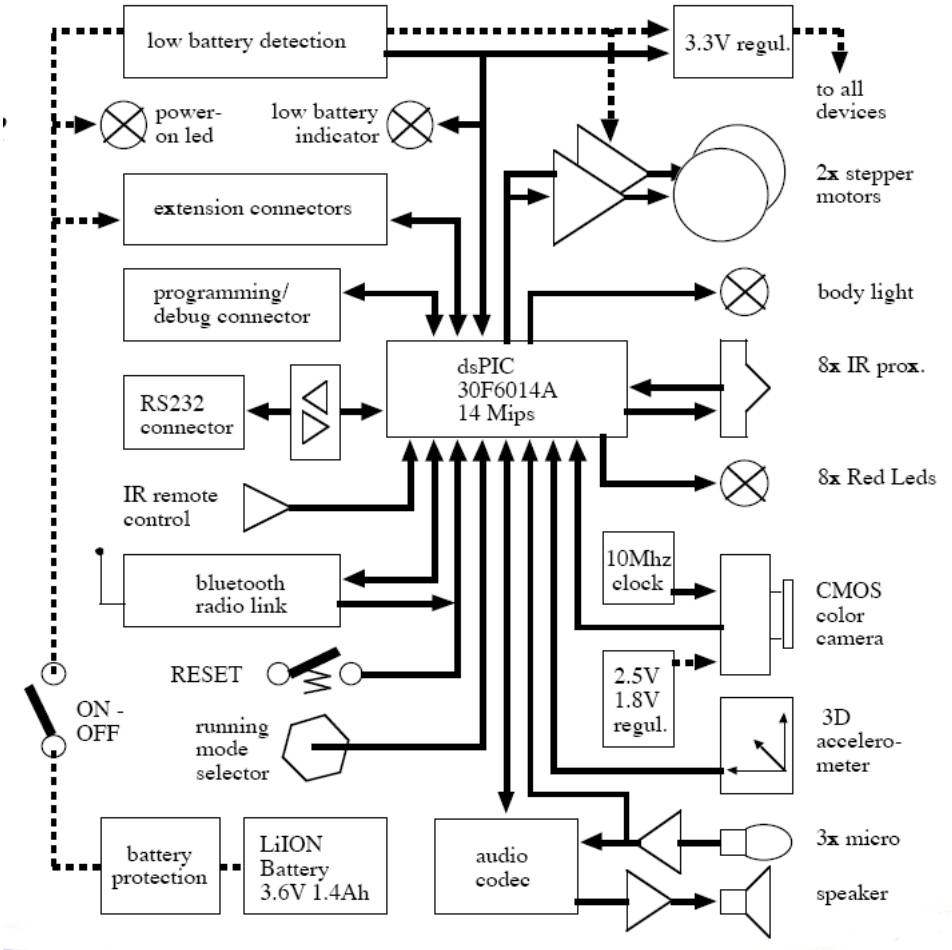


Figure 16: Schematic diagram of the electronic circuit

As it can be seen in the diagram, the core of the robot lies in the micro-controller. For this reason a very efficient digital signal processor is required. The Digital Signal Programmable Interface Circuit by the Microchip company is able to do this while at the same time its price respects the budget constraints.
The camera has a 640x480 color pixel resolution—unfortunately, the micro-controller's memory is limited to handling a max resolution of 40x40 pixels at 4 frames-per-second[5]. At the same time this limit remains sufficient for the prototyping nature of the e-puck.

With multiple sensors, one powerful micro-controller, and affordable actuators—the e-puck becomes the ideal robot for learning and experimenting in mobile robotics.

Together with the Webots simulation environment. The pair has the potential to become a standard tool in mobile robotics education and research settings

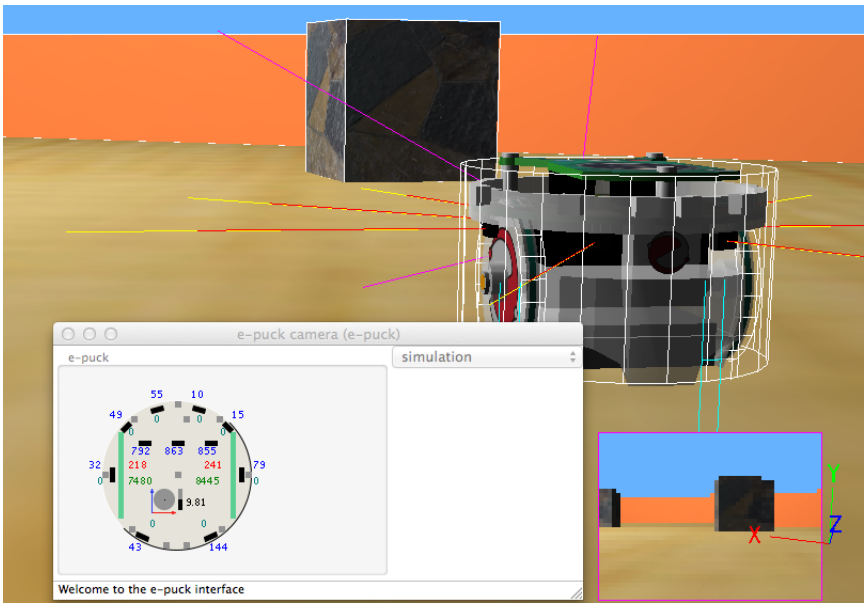


Figure 17: The e-puck's sensor data in visualized in *Webots*

Part II

CHALLENGE

3

THE RAT'S LIFE PROGRAMMING CONTEST

The Rat's Life Programming Contest was developed as an initiative by the ICEA project [1] as a benchmark to measure research results and to stimulate interest in bio-inspired robotics. Although the *rats* fought for survival from January 7th to June 30th 2010, one simulation *round* is still run every week for entertainment purposes [8].

3.1 OVERVIEW

Two simulated e-puck *rats* compete for energy resources in a maze-like environment. They must use effective algorithms to identify the navigating path (and state) of the battery feeders in order to use them at the most appropriate time. Their battery is constantly decreasing as the time passes, and the first one to run out of battery will lose the match.

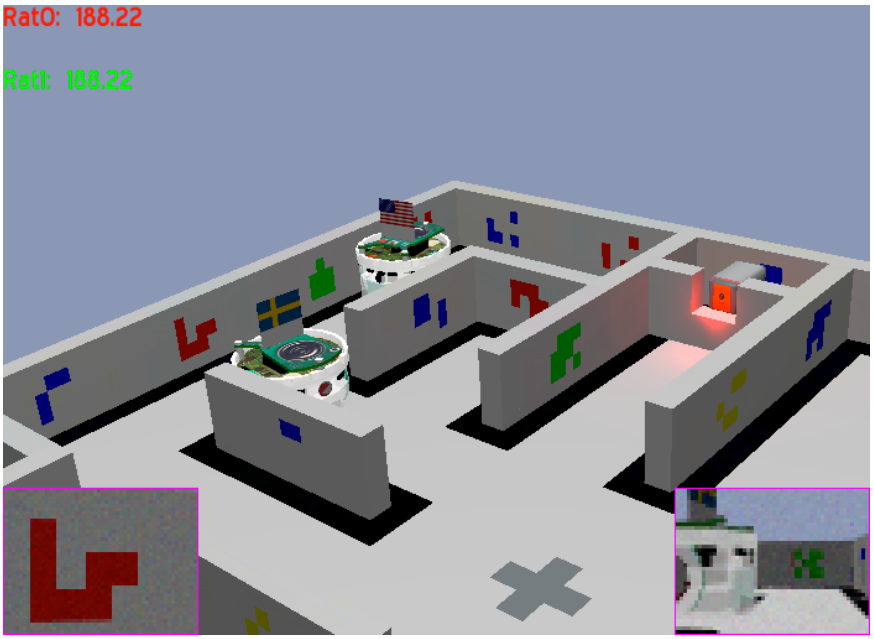


Figure 18: A simulation model of the Rat's Life Programming Contest

1. "Integrating Cognition, Emotion and Autonomy" was a four year project which ended in 2010. Its primary aim was to develop a cognitive systems architecture based on the physiology of the mammalian brain. It was funded by the Cognition Systems and Robotics unit of the European Commission.

3.2 CONTEST GUIDELINES

3.2.1 *Round guidelines*

New rats enter a *round* with the lowest possible ranking. A *round* consists in a set of matches. In each *match* two rats fight against one another. The lowest-ranking rat fights against the second lowest-ranking rat; the winner of this match will fight with the third lowest-ranking rat, and so on until the second best-ranking rat fights against the best-ranking rat—the winner of this match will become the best-ranking rat.
Considering the wide range of solution algorithms, and the role that luck plays in a *match*, this type of ranking guarantees that:

- A rat can climb to the best ranking position in one round.
- A rat can not lose more than one ranking position per round.

This method encourages the use of more general algorithms that are able to survive against all kinds of opposing strategies. The contestants will mark their ranking position over time; eventually, if there are no new contestants the ranking positions will settle. In this way one round is analogous to one iteration of the bubble-sort algorithm.

3.2.2 *Match guidelines*

3.2.2.1 *Winner*

The last robot to run out of energy wins.

3.2.2.2 *Energy [8]*

- An e-puck has between 0 to 200 units of energy.
- At the beginning of the match, an e-puck has 200 units of energy.
- The first robot having 0 unit of energy loose the game.
- A feeder has between 0 and 100 units of energy.
- At the beginning of the match, a feeder has 100 units of energy.
- The time for an e-puck which doesn't move to pass from 100 to 0 units of energy is 90 seconds. Moving the e-puck gives a very small advantage in term of energy, in order to encourage displacements.
- The time for a feeder to pass from 0 to 100 units of energy is 120 seconds. When a feeder has 100 units of energy, its LED is switched on. Otherwise the LED is switched off.
- When a feeder has 100 units of energy and when an e-puck is placed in front of this feeder (6.4cm around the center of the feeder), the energy of the feeder and of the e-puck follow this rule:

```
if (FeederEnergy==100 and distance(EPuck, Feeder) < 6.4[cm]):
  if (EPuckEnergy < 100):
    EPuckEnergy = MINIMUM(100, EPuckEnergy+100)
    FeederEnergy = 0
  else:
    FeederEnergy = 0
```

3.2.2.3 *Maze [8]*

- The maze is composed of 10x10 square cells surrounded by an external wall.
- An internal wall can split two neighbouring cells.
- All the walls are white, and are surrounded with two grey intervals.
- The internal walls are placed according to the *Randomized Prim's algorithm*. This implies that every cell is accessible from every other cell.
- Once this algorithm applied, 20% of the walls are removed randomly in order to create more pathes between cells.
- A maze has 4 feeders.
- The feeders are placed randomly over the cells having exactly three walls.
- A landmark is a drawing over the wall, representing a matrix of 4x3 pixels. A pixel has either the color of the landmark (red, green, blue or yellow), or either the color of the wall.
- A landmark is placed on each side of each wall.
- All the tiles on the ground are white except the tiles around the walls which are black.

3.2.3 *Technical guidelines*

The simulation is run on a computer equipped with [8]:

- Hardware: Intel Core2 Quad Q6600 2.4GHz processor, nVidia GeForce 8500GT graphics card.
- Software: Ubuntu 10.04 64bits Linux with up-to-date nVidia drivers installed, Java 1.6.0_18, Webots 6.3.1.

The organizers do not guarantee that the simulation will behave in the same way as in the contestant's computer.

Robots that attempt to do any of the following will be immediately disqualified:

- Display a Graphical User Interface
- Read input from a hardware device
- Use the network
- Play sound
- Perform I/O operations outside of the file's containing directory

A single file named `Rat0.jar` should be uploaded to the contest's website. This file can be generated by selecting the `Rat0.java` file in the built-in text editor and using the *Make JAR file* feature from the *Build* menu item.

3.3 REAL WORLD

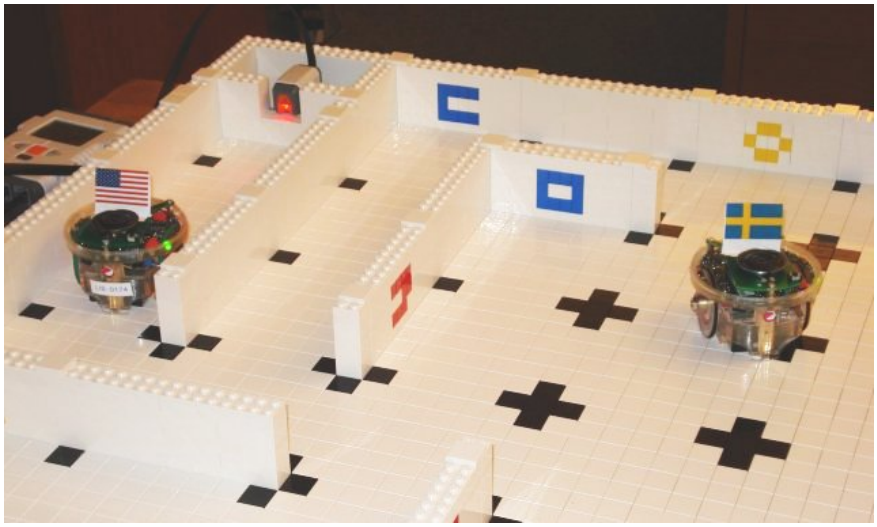


Figure 19: A real model of the Rat's Life Programming Contest

The simulated environment was inspired by a real world setup that was built using Lego© bricks and different parts from the affordable Lego© Mindstorms© robotics kit. The official website of the contest provides the instructions used to build this replica. Nevertheless, the authors claim that the instructions are out-dated and that they are not guaranteed to work as expected.

4 ALGORITHMS

Before attempting to understand any algorithm; it is important to correctly understand the problem that it is trying to solve. I have ignored the parts of the problem that are trivial to facilitate comprehension.

4.1 DISSECTING THE PROBLEM

4.1.1 *Anatomy of autonomous robots*

$$\text{Sensors} \rightarrow \text{Controller} \rightarrow \text{Actuators}$$

The sensors *sense* the environment, and the actuators *act* on the environment. The environment changes and the sensors *sense* different values after the actuators have *acted*. At the same time, external controllers may also *act* on the same environment.

4.1.2 *Anatomy of a rat*

In the Rat's Life contest there are ten sensors available to the controller: 8 infra-red sensors, 1 camera sensor, and 1 accelerometer sensor; the available actuators are 2 stepper motors, and 9 LED's. The accelerometer and the LED's are superfluous for the contest requirements.

4.1.3 *The environment*

In the context of an autonomous robot it is more meaningful to separate the description of the environment into *static* and *dynamic* elements. A map can only be built based on the static elements and a map will be misleading if some *dynamic* elements are miscategorized as *static*.

4.1.3.1 *Static elements*

Every wall in the maze has a unique landmark that can be used to build a map.

4.1.3.2 *Dynamic elements*

If a rat touches a feeder then the feeder will switch off until it is fully recharged. The opponent is navigating freely around the maze.

4.1.4 *Requirements*

Recognize feeders and identify a path leading to them at all times. Define a strategy for optimal feeder use.

4.2 SAMPLE ALGORITHM

The algorithm provided with *Webots* does not identify paths to the feeders since this problem involves recognizing the static elements, building a map with those elements, and locating the robot within that map. The solution to those problems is complex and the solving them is up to the contestant.
Nevertheless, a simpler algorithm is provided as a starting point. It avoids obstacles, turns when the robot is in front of a wall, and goes towards a feeder if it gets close enough to the camera.

4.2.1 *General Behavior*

The following is a simplified version of the sample algorithm included with *Webots*. The LED's used to signal turning or a succesful recharge have been ignored. Note that a used feeder will remain switched off until it has been completely recharged.

Algorithm 1 Main loop

```
repeat {one time step}
  image ← read camera
  surroundings ← read distance sensors
  if obstacles in surroundings then
    avoid them
  end if
  if feeder in image then
    get closer
  end if
until controller is killed
```

The e-puck's behavior can be thought of as a magnetic hockey puck that repulses walls, and gets attracted to feeders. It uses the IR sensors to repulse obstacles that get too close (by changing its direction accordingly), and to turn if there is an obstacle facing the front. The camera looks for feeder pixels and goes towards them (by changing its direction). When the e-puck is not changing it's direction, it is rotating both motors at maximum speed.

4.2.2 *Staying on the right path*

The navigation algorithm is composed of both *proximal* and *distal* based controllers:

4.2.2.1 *Collision Avoidance*

Proximal (or reactive) robot control is achieved through *hard-wiring* the sensors directly to the actuators without any kind of reasoning *per-se*. The *wires* may have different weights that determine how relevant that wire's input should be in relation to the other wires which are connected to the same acting end.

In our case the collision avoidance algorithm uses the *proximal* paradigm.

Algorithm 2 Collision avoidance

```
left speed ← max speed
right speed ← max speed
for all sensors do
  left speed = left speed − ((slow motion weight + collision avoidance weight) ∗ sensor value)
  right speed = left speed − ((slow motion weight − collision avoidance weight) ∗ sensor value)
end for
```

4.2.2.2 *Obstacle avoidance*

Distal robot control is based on conditional statements (i.e. rules). For example, a rule can be that when our robot is confronted with an obstacle (e.g. a wall, a dead feeder, or the other robot); he should turn around until his path is clear again.

Algorithm 3 Obstacle avoidance

```
if obstacles in front sensors then
  if turning is stopped then
    turning = right or left {Chosen randomly}
  else if turning is left then
    left speed = min speed
    right speed = max speed
  else if turning is right then
    left speed = max speed
    right speed = min speed
  end if
else
  turning = stopped
end if
```

4.2.3 *Going towards the feeders*

Our robot detects the feeders and calculates the required rotation to point towards them (if necessary). Detecting a blob, in this case, is simply a matter of looking for the right pixels. The values of the feeder pixel colors were empirically defined.

Algorithm 4 Feeder detecting

```
blob ← 0
counter ← 0
for y = image height/3 to 2∗image height/3 do
  for x in image width do
    if pixel(x, y) is feeder pixel then
      blob = blob +x
      counter = counter +1
    end if
  end for
end for
```

Algorithm 5 Feeder following

```
Require: counter > 2 {Feeder found}
blob = blob / counter {Average}
dx = blob − ((camera width /2) * 10)
if dx ≠ old dx then {Still need to adjust direction}
  left speed ← 2 ∗ dx
  right speed ← −2 ∗ dx2
  old dx = dx
else {Go straight slowly}
  left speed ← max speed /3
  right speed ← max speed /3
end if
```

4.3 IMPROVEMENTS

There are only a few modifications that can be made to improve the existing algorithms without completely changing them. Those are:

- Using the front left and front right sensors separately to decide wether to turn left or right.
- Fine tuning the turning algorithm used when following a feeder to minimize speed reduction
- Introduce a noise factor in both obstacle avoidance algorithms so that the robot doesn't get locked in a T-shaped corridor.

5 CONCLUSION

5.1 RELEVANCE

After thoroughly exploring the simulation software, and the contest environment; we observed that software simulation tools can significantly reduce costs in both time and money. We also analyzed the autonomous navigation problem; as well as the most basic algorithms used for autonomous mobile robot navigation, and blob detection. Hopefully this work can help improve the learning curve of enthusiastic students.

5.2 FUTURE WORK

After conquering the basics, the interested students should focus on understanding more advanced concepts in robotics as they can help design a significantly better algorithm. They can use the contest to test their knowledge as they progress. Recommended topics are: locomotion, kinematics, path-planning, odometry, control theory, fuzzy logic, computer vision, edge detection, feature detection, feature extraction, Bayesian probability, artificial intelligence, Markov localization, and SLAM. They should extensively document their learning to help other novice students.

Part III

APPENDIX

APPENDIX

A.1 SOURCE CODE

The sample source code povided with *Webots* refactored into a more meaningful structure (the algorithm remains unchanged):

```
import com.cyberbotics.webots.controller.Accelerometer;
import com.cyberbotics.webots.controller.Camera;
import com.cyberbotics.webots.controller.DifferentialWheels;
import com.cyberbotics.webots.controller.DistanceSensor;
import com.cyberbotics.webots.controller.LED;
import com.cyberbotics.webots.controller.LightSensor;

import javax.swing.*;
import java.awt.*;
import java.awt.image.*;
import java.util.Random;

public class Rat0 extends DifferentialWheels {

    protected final int timeStep = 32;
    protected final boolean gui = true;
    protected final double maxSpeed = 300;
    protected final double[] collisionAvoidanceWeights =
        {0.06,0.03,0.015,0.0,0.0,-0.015,-0.03,-0.06};
    protected final double[] slowMotionWeights =
        {0.0125,0.00625,0.0,0.0,0.0,0.0,0.00625,0.0125};

    protected Accelerometer accelerometer;
    protected Camera camera;
    protected int cameraWidth, cameraHeight;
    protected DistanceSensor[] distanceSensors = new
        DistanceSensor[8];
    protected LightSensor[] lightSensors = new LightSensor[8];
    protected LED[] leds = new LED[10];

    private class State {

        public int blink, oldDx, blobX, blobY, blobCounter;
        public boolean turn, right;
        public double battery, oldBattery, leftSpeed, rightSpeed;
        public Random r;
        public int image[];
        public double distance[] = new double[8];
        public int ledValue[] = new int[10];
```

```
    public State() {
        this.blink = 0;
        this.oldDx = 0;
        this.r = new Random();
        this.turn = false;
        this.right = false;
        this.battery = -12345; // not initialized
        this.oldBattery = -1.0;
        this.image = null; // not initialized
        this.distance = new double[8];
        this.ledValue = new int[10];
        this.leftSpeed = maxSpeed;
        this.rightSpeed = maxSpeed;
        this.blobX = 0;
        this.blobY = 0;
        this.blobCounter = 0;
    }
}

public Rat0() {
    accelerometer = getAccelerometer("accelerometer");
    camera = getCamera("camera");
    camera.enable(8*timeStep);
    cameraWidth = camera.getWidth();
    cameraHeight = camera.getHeight();
    for (int i = 0; i < 10 ; i++) {
        leds[i] = getLED("led" + i);
    }
    for (int i = 0; i < 8; i++) {
        distanceSensors[i] = getDistanceSensor("ps" + i);
        distanceSensors[i].enable(timeStep);
        lightSensors[i] = getLightSensor("ls" + i);
        lightSensors[i].enable(timeStep);
    }
    batterySensorEnable(timeStep);
}

public static void main() {
    Rat0 rat0 = new Rat0();
    rat0.run();
}

public void run() {

    State now = new State();
```

```
    do { // One step of the simulation

        readSensors(now);
        setAvoidObstacles(now);

        boolean feederDetected = detectFeeder(now);
        if (feederDetected) setFollowFeeder(now);

        booleand batteryRecharged = checkRecharge(now);
        if (batteryRecharged) setBodyLED(now);

        setBlinkingBackLED(now);
        setActuators(now);

        // Act while the controller has not been killed
     } while (super.step(timeStep) != -1);
}

private void readSensors(State now) {
    now.image = camera.getImage();
    for (int i = 0; i < 8; i++) {
        now.distance[i] = distanceSensors[i].getValue();
    }

    now.battery = batterySensorGetValue();
    for (int i = 0; i < 10; i++) {
        now.ledValue[i] = 0;
    }
}

private void setAvoidObstacles(State now) {
    // Obstacle avoidance behavior
    now.leftSpeed  = maxSpeed;
    now.rightSpeed = maxSpeed;
    for (int i = 0; i < 8; i++) {
        now.leftSpeed  -= (slowMotionWeights[i] +
            collisionAvoidanceWeights[i]) * now.distance[i];
        now.rightSpeed -= (slowMotionWeights[i] -
            collisionAvoidanceWeights[i]) * now.distance[i];
    }

    // Return either to left or to right when there is an
        obstacle
    if (now.distance[6] + now.distance[7] > 1800 || now.
        distance[0] + now.distance[1] > 1800) {
        if (!now.turn) {
            now.turn = true;
            now.right = now.r.nextBoolean();
        }
        if (now.right) {
            now.ledValue[2] = 1;
            now.leftSpeed  =  maxSpeed;
```

```
            now.rightSpeed = -maxSpeed;
        } else {
            now.ledValue[6] = 1;
            now.leftSpeed  = -maxSpeed;
            now.rightSpeed =  maxSpeed;
        }
    } else {
        now.turn = false;
    }
}

private boolean detectFeeder(State now) {
    now.blobX = 0;
    now.blobY = 0;
    now.blobCounter = 0;
    for (int x = 0; x < cameraWidth; x++) {
        for (int y = cameraWidth / 3; y < 2 * cameraWidth /
            3; y++) {
            int pixel = now.image[y * cameraWidth + x];
            if (Camera.pixelGetGreen(pixel) >= 248 &&
                Camera.pixelGetBlue(pixel) >= 248) {
                now.blobX += x;
                now.blobY += y;
                now.blobCounter++;
            }
        }
    }

    return now.blobCounter > 2; // Significant enough
}

private void setFollowFeeder(State now) {
    now.blobX /= now.blobCounter;
    now.blobY /= now.blobCounter;
    int dx = (now.blobX - cameraWidth / 2) * 10;
    if (dx > 0) now.ledValue[1] = 1;
    else now.ledValue[7] = 1;
    if (now.oldDx != dx) {
        now.leftSpeed  =  2*dx;
        now.rightSpeed = -2*dx;
        now.oldDx = dx;
    } else {
        now.leftSpeed  = maxSpeed / 3;
        now.rightSpeed = maxSpeed / 3;
    }
}

private void checkRecharge(State now) {
    boolean batteryRecharged = now.battery > now.oldBattery;
    now.oldBattery = now.battery;
    return batteryRecharged;
}
```

```
    private void setBodyLED(State now) {
        now.leftSpeed  = 0.0;
        now.rightSpeed = 0.0;
        now.ledValue[8] = 1; // Body LED
    }

    private void setBlinkingBackLED(State now) {
        now.blink++;
        if (now.blink >= 20) {
            now.ledValue[4] = 1; // Back LED
            if (now.blink == 40) now.blink = 0;
        }
    }

    private void setActuators(State now) {
        for (int i = 0; i < 10; i++) {
            leds[i].set(now.ledValue[i]);
        }
        super.setSpeed(now.leftSpeed, now.rightSpeed);
    }
}
```

REFERENCES

[1] Anonymous. Google cars drive themselves. *Informationweek*, October 2010. Retrieved from ABI/INFORM Global.

[2] F Comment Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli. The e-puck, a robot designed for education in engineering. *International Journal of Advanced Robotics Systems*, pages 59–65, 2009.

[3] Cyberbotics. About webots. http://www.cyberbotics.com/about, 2012. Accessed: 01/04/2012.

[4] Cyberbotics. Documentation. http://www.cyberbotics.com/documentation, 2012. Accessed: 01/04/2012.

[5] Cyberbotics. e-puck. http://www.cyberbotics.com/e-puck, 2012. Accessed: 01/04/2012.

[6] Cyberbotics. Features. http://www.cyberbotics.com/features, 2012. Accessed: 01/04/2012.

[7] Cyberbotics. Users. http://www.cyberbotics.com/users, 2012. Accessed: 01/04/2012.

[8] Rat's Life. Rat's life. http://www.ratslife.org/, 2012. Accessed: 01/04/2012.

[9] O. Michel. Webots: Professional mobile robot simulation. *International Journal of Advanced Robotics Systems*, pages 39–42, 2004.

[10] Sebastian Thrun. A personal account of the development of stanley, the robot that won the darpa grand challenge. *AI Magazine*, January 2006. Retrieved from ABI/INFORM Global.

DECLARATION

I hereby declare that this thesis is my own work.

Brussels, Belgium, January 2012

Mauricio Zepeda