BACHELORARBEIT

Sensor Extensions for the Rossum's Playhouse Robot Simulator

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich
Institut für Technische Informatik 182

durchgeführt von

Laszlo Keszthelyi
Matr.-Nr. 0025953
Lessinggasse 18, A-2231 Strasshof

Wien, im September 2007 .....

Sensor Extensions for the Rossum's Playhouse Robot Simulator

When developing software for mobile-robots it is not always possible to test them in reality. Therefore simulators of robots and their enviroment are very useful to test the software and find bugs before running it on real hardware. Furthermore robot-simulators are practical for scientific research, e. g. developing neural networks.
The main aim of this thesis is to analyze available open-source robot-simulators for their usability as a testbed for intelligent control strategies. This thesis concentrated on open-source simulators, because of the cost advantage, the avoidance of licensing issues and the possibility to extend or adapt the sources to our requirements.
This report starts with a short introduction to robotics and simulations in general befor it concentrates on the comparison of the diffent robot simulators. Finally the solution of three required extensions of *Rossum's Playhouse*, that had to be implemented within the scope of this thesis, will be presented.

Contents

1 Introduction

Writing software for robots is not only challenging but also difficult in testing. For example when we want to test a large number of variants, we have to program and test the robot probably a thousand times. Analysing or identifying errors directly on the robot can get difficult, because most of them only provide visual feedback. In some cases we might first want to know if our software runs without any unexpected sideeffects because the robot hardware is valuable and we don't want to damage it. Therefore simulators of robots and their enviroment are good tools to test our software before running it on real hardware. Simulations allow us to study the behavior of our mobile robot in a simplified version of the real world. Depending on the requirements the simulation enviroment be a simple 2D-map where the robot is able to move around or a wide more complex including a physics-engine and a 3D world.
There are many robot simulators open-source and commercial but not all of them are applicable in scientific research because we might have to adapt the simulator to our needs. For this reason we are analysing robot simulators for their suitability in scientific research especially as a testbed for intelligent control strategies (i. e. developing neural networks) in this thesis. However, primarily we will focus on open-source solutions because they are free, the sources are available for everyone and hence customisable. Although we will take a short look on commercial software solutions even if their sources are not available and adaptation to meet special requirements is not possible.
Sometimes it is required to extend or modify the existing simulation environment to meet the requirements or gain a better performance. In this thesis we describe three requirements that had to be met by the *Rossum's Playhouse* robot simulator and how we solved them.

1.1 Structure of the Thesis

Chapter 2 is an overview about robotics and simulation, including a brief history of robotics. In chapter 3 we present our results of the analysis of different available robot simulators. Section 3.1 is about open-source robot simulators and their basic abilities. In Section 3.2 is a an overview about two commercial robot simulators. Section 3.3 compares the introduced simulators to each other

and analysed for their usability as a testbed for intelligent control strategies.
The chapters 4, 5 and 6 are about the three requirements that had to be solved in the scope of this thesis:

- Synchronising Client-Server
- Implement a Pollable-Sensor
- Implement a Pivotale-Sensor

Chapter 4 is an introduction to the three problems, chapter 5 will describe the implementation in detail and finally chapter 6 will show how the results have been affected by our modifications.
At last, chapter 7, which is the end of this thesis, summarises the results of the presented work.

2 Fundamentals and Concepts

As we mentioned before, building a robot is not easy because it involves skills from many disciplines (hardware design, embedded firmware, sensor selection, mechanical design,...). Therefore simulation enviroments are very practical because they can provide a virtual "arena" for testing, measuring and visualizing robotics algorithms without high cost and time of development and the risk to damage the robot while testing.
Before we start with our comparison of the different open source robot simulation enviroments we want to give you a short introduction (only the basics are discussed here because they are needed for the comparison later everything else would go beyond the scope of this thesis) to robotics in section 2.1 and simulation in section 2.2.

2.1 Robotics

Robotics is the field of computer science (i. e. artificial ingteligence) and engineering that involves conception, design, manufacture, and operation of robots, devices that can move and react to sensory input.
Even if many people think of a physical robot when they here the word "robot" it can refer to both physical robots and virtual software agents (rather known as "bots"). In this thesis whenever we talk about "robots" we mean physical robots.
Robots may have different appearance and their capabilities vary vastly but, all robots share the features of a mechanical, movable structure under some form of autonomous control.

2.1.1 Autonomous Robots

Autonomous robots are robots which can perform desired tasks without continuous human guidance or intervention in unstructured enviroments. Robots differ in the degree of autonomy, for example a high degree autonomy is desired in fields such as space exploration, where communication delays are unavoidable. But there are also mundane uses which benefit from having some level of

autonomy, like cleaning floors, mowing the lawns, and waste water treatment. A fully autonomous robot has the ability to

- Gain information about the environment.
- Work for an extended period without human intervention
- Move either all or part of itself throughout its operating enviroment without human assistance.
- Avoid situations that are harmful to people, property, or itself

An autonomous robot may also learn or gain new capabilities like adjusting strategies for accomplishing its task(s) or adapting to changing surroundings. Although autonomous robots should be able to do lots of things on their own, they are just machines and as machines they require regular maintenance. This is done by humans but in the future robots will repair robots.

2.1.2 Configuration of a Robot

To be able to gain information about the environment, or to interact with it, a robot needs a set of sensors, a set of effectors and a control system that allows the robot to act in an intentional and useful way. These are the 3 main elements of a robot and they act together in a closed loop. Sensors are feeding the control system with information about the enviroment and the control system specifies the actions of the effectors to effect changes in the enviroment (or simply just move itself around). The sensors than can validate this changes and feed back the new state of the enviroment to the control system.

Sensing the Enviroment

Sensors are transducers which convert one type of energy or a signal of some sort into data that the robot can understand and act accordingly. Depending on the application of the robot, different sensors are used for sensing the enviroment. The most common sensors are:

- mechanicle (e. g. pressure sensor, switch, flow sensor, ...)
- acoustic (e. g. microphones, hydrophones, ...)
- thermal (e. g. thermometers, calorimeter, ...)
- electromagnetic (e. g. multimeter, metal detectors, ...)
- optical radiation (e. g. infra-red, photocells, ...)
- motion (e. g. tachometer, turn coordinator, ...)
- distance (e. g. infra-red, sonar, ...)

Interacting with the Enviroment

To be able to interact with its enviroment a robot needs some kind of effector (also called actuator). The type depends on the application of the robot but almost all are mechanic. The most common effectors are:

- pneumatic actuators
- electric actuators
- motors (most mobile robots are using electromotors and some a combustion engine)
- hydraulic cylinders

2.2 Simulation

A simulation is a method which models key-characteristics of real-life or a hypothetical situation so that it can be studied in a safe enviroment. By the use of simulations it is possible to make predictions about the modeled system in different situations. Simulations are not limited to simplify or allow studies and predictions, they can also be used for education and training. This usuallly occures when it is prohibively expensive or too dangerous to allow trainees to use the real equipment in the real world.
Another type of simulators are the emulators, that are used to execute programms in a controlled testing enviroment. The programmer is able to control the speed and execution of the simulation and has access to all informations of the simulated operations.

2.2.1 Robots and Simulation

In the field of robotics, simulation plays a key role because it permits experimentation that would otherwise be expensive and/or time-consuming. It also allows the evolution of robotics control systems, which depend on random permutations of the control system over many generations (e. g. genetic algorithms).
Another great advantage is the simulation of mulitiple robots at the same time. A popular venue for these simulations is in Robot Soccer, where either through simulation or with physical robots, one team of robots competes against another, making it a challenging test of cooperative robot behavior.
The downside of simulations is that a simulation is always an abstract model of the real environment that disregards particular aspects (for example, many models do not take failing or noisy sensor into account). Often, also several

parameters from the real-world scenario are unknown, so that it is difficult to define a valid simulation. The simulation model must be carefully chosen (depending on the purpose and real-world situations), otherwise the results from simulation do not hold for the intended purpose. Despite the disadvantages, simulations are very helpful for research and development.

3 Overview on Robot Simulation Software

In this chapter we will introduce and compare different robot simulators. First we will have a look at some open-source simulators not only because they are free to everyone but rather because their sources are public and therefore these simulators are better customisable to specific needs of a particular purpose. After that we will have a look at commercial simulators. Finally we will present a comparison between the robot simulators.

3.1 Open-Source Simulators

As we mentioned in this chapter's introduction, open-source simulators are very attractive free to everyone and their sources are public, allowing the extension, modification and adaption of the programme by everyone. It also allows a better insight into how the simulator works and therefor making it possible to write better simulations of a particular situation. The following four robot simulators are some of the most popular open-source simulators.

3.1.1 Rossum's Playhouse

The Rossum's Playhouse[1] is a client/server-based robot simulator written in Java. It is an 2D robot simulator intended to aid developers implementing control and navigation logic. The Rossum's Playhouse allows to build a data-configurable robot which can interact with a simulated landscape or solve a virtual maze. For a better understanding how to create simulations for the Rossum's Playhouse a user-guide is also available. It not only contains a description of the simulator and examples it also contains a short introduction to Java, which is helpful to those that are not familiar with the language.
The communication between client and server is based on TCP/IP-sockets, which allows to write clients in a different language than Java. The server represents the simulation engine and the simulation environment (floor plan).

[1] rossum.sourceforge.net

The client connects to the server uploads the details of robot and starts the simulation. The simulated robot is controlled by the client, this is achieved by sending sensor and positioning informations continuously from the server to the client. The server is designed to accept connections from multiple clients simultaneously but it does not support interactions between robots yet.
Although the Rossum's Playhouse is lacking a physics engine it could be possible to write an extension for it, allowing to move objects around or simulate the behaviour of the robot actuators in a more realistic way.

3.1.2 Player/Stage/Gazebo

As the Rossum's Playhouse robot simulator the Player/Stage/Gazebo[2] is a client/server-based simulator but it is written in C/C++. It also differs in its architecture because it consists of a main programme, "Player", and two plugins, "Stage" and "Gazebo".

Player

Player is a network server for robot control. Running on a robot, Player provides a simple interface to the robot's sensors and actuators over the IP network. A client program connected to Player can read data from sensors, writing commands to actuators, and configure devices on the fly. Player supports a variety of robot hardware and it is possible to add support for new hardware.

Stage & Gazebo

The Player/Stage project provides two multi-robot simulators: Stage and Gazebo. Since Stage and Gazebo are both Player-compatible, client programs written using one simulator can usually be run on the other with little or no modification. The key difference between these two simulators is that whereas Stage is designed to simulate a very large robot population with low fidelity (fairly simple, computationally cheap models of lots of devices), Gazebo is designed to simulated a small population with high fidelity (realistic sensor feedback and physically plausible interactions between objects including an accurate simulation of rigid-body physics). Another difference between them is that Stage uses a 2D environment and Gazebo a 3D Environment for the simulations.

[2] playerstage.sourceforge.net

3.1.3 Simbad robot simulator

Simbad [3] is a Java-based 3D multi-robot simulator developed for scientific and educational purposes. The main motivation is to provide an easy-to-use all-in-one package for Evolutionary Robotics. The Simbad package includes a complex 3D simulation engine (Simbad), a Neural Network library as well as a complete Evolutionary Algorithm library (Genetic Algorithm, Evolutionary Strategy, Genetic Programming with trees or graphs).

The Simbad Simulator

Simbad is a Java-based 3D multi-robot simulator developed for scientific and educational purposes. According to Simbad's description it allows the application of batch simulation in the context of heavy computation for learning (e. g. evolutionary algorithms). The robot simulator comes with an integrated simple physical engine and allows Python scripting with jython[4].

Neural Network Library

The PicoNode library provides a general graph-based representation framework along with two implementations: feed-forward and recurrent neural networks. The use of PicoNode is not limited to robot control; it has been designed so as to ease building of simple (e. g. multi-layered perceptrons) as well as less simple (e. g. N-layers recurrent nets) neural networks.[5]

Evolutionary Algorithm Library

PicoNode provides standard supervised learning algorithm (e. g. Back-Propagation), however such learning algorithms are usually of little use in the context of sparse, noisy, delayed and asynchronous reinforcement signals that are usually part of the task of control learning in mobile robotics (e. g. a binary reward (success/failure) is provided only when the robot may reach the exit of a maze).[4]

[3] simbad.sourceforge.net

[4] www.jython.org

[5] taken from "Simbad : an Autonomous Robot Simulation Package for Education and Research" (www.lri.fr/ bredeche/MyBiblio/SAB2006simbad.pdf)

3.1.4 c't-Sim

The c't-Sim is a client/server-based multi-robot simulator written in Java for the c't-Bot (a small robot programmed in C), both developed by German computer magazine c't[6]. The simulator includes a maze-generator allowing the generation of random floor-plans without great effort. It is possible to run c't-Bot control programs on the simulator by first compiling and than running the programm as a tcp-client that connects to a running ct-Sim[7]. The sources are well documented and a FAQ helps to find the most common problems when trying to run the simulator. All in all the c't-Sim is a simple simulator but it can be extended to a more complex one that could support various robot types. It does not support a physics engine.

3.2 Commercial Simulators

On the one hand commercial simulators are in the majority of cases continuously improved but on the other hand it is often not possible or not permitted to write own extensions or to modify the simulator. For that reason the simulator's sources are very rarely public. We will now examine two well-known commercial simulators.

3.2.1 Microsoft Robotics Studio

The Microsoft Robotics Studio[8] is a Windows-based environment for robot control and simulation for hobbyist, academic and commercial developers to create robotics applications for a variety of hardware platforms.[9]
It comes with a 3D simulation environment and includes PhysX[10] as its physics-engine. It can be also used for other 3D physical simulations than just for simulating robot behaviour. The robots are mainly programmed via the Microsoft Visual Programming Language[11] that takes getting used to, but it is possible to write programs in other languages such as C# and Visual Basic .NET, JScript, and IronPython.

[6] www.heise.de/ct/projekte/machmit/ctbot/wiki
[7] wiki.ctbot.de/index.php/C%27t-Bot_Simulation
[8] msdn.microsoft.com/robotics
[9] taken from msdn2.microsoft.com/en-us/library/bb483024.aspx
[10] www.ageia.com
[11] msdn2.microsoft.com/en-us/library/bb483088.aspx

As previously mentioned the Microsoft Robotic Studio supports a wide variety of robots:

- CoroWare's CoroBot[12]
- Lego Mindstorms NXT[13]
- Robosoft's robots[14]
- Boe-Bot[15]
- iRobot Create[16]

Although the sources are not public the Microsoft Robotic Studio includes support for packages to add other services to the suite. One currently available is a community-developed Maze Simulator. Like the maze-generator for the c't-Sim it creates worlds with walls that can be explored by a virtual robot.

3.2.2 Webots

Webots[17] is a 3D multi-robot simulator similar to Simbad and Gazebo. It has an integrated physics engine and allows the modelling of various robots because it includes a wide range of different loco-motions, actuators and sensors. A built-in IDE helps you programming the robot, but you can also use your favorite development environment. The programme can than be simulated or transfered directly to a real robot. Webots supports roughly the same real-robots as Microsoft Robotics Studio does. It's main disadvantage is its high cost, because there are open-source simulators which also provide among other things a 3D enviroment, a physics-engine and a library of different robots and their components.

[12] www.corobot.net
[13] mindstorms.lego.com
[14] www.robosoft.fr/eng/
[15] en.wikipedia.org/wiki/Boe-Bot
[16] www.irobot.com
[17] www.cyberbotics.com/products/webots

3.3 Comparison

Simulator	Rossum's Playhouse	Player Stage/Gazebo	Simbad robot simulator
Language	Java	C/C++	Java
Platforms	Windows, Linux, Mac OS X, Solaris	Windows, Linux, Mac OS X, Solaris	Windows, Linux, Solaris
Price	free[1]	free[1]	free[1]
Field of Application	Research, Development, Education, Hobbyist	Research, Development, Education, Prototyping, Hobbyist	Research, Education, Hobbyist
Release	0.61	1.4	2.1.0rc1
Sources available	yes	yes	yes
Architecture	Client/Server	Client/Server	Standalone
Multi-Robots	planned	supported	supported
Extendable	yes	yes	yes
Physics-Engine	none	built-in	ODE[2]

[1]Open-Source
[2]Open Dynamics Engine (www.ode.org)

Simulator	c't-Sim	Microsoft Robotics Studio	Webots
Language	Java	Microsoft Visual Programming Language	C/C++, Java
Platforms	Windows, Linux	Windows	Windows
Price	free[1]	free[2]	Pro: SFR 3450,- Pro (academic): SFR 2300,- Edu: SFR 320,-
Field of Application	Development, Hobbyist	Research, Development, Education, Prototyping, Hobbyist	Research, Development, Education, Prototyping
Release	13	1.5	5
Sources available	yes	no	no
Architecture	Client/Server	Client/Server	Standalone
Multi-Robots	supported	supported	supported
Extendable	yes	yes	no
Physics	none	Ageia PhysX	ODE[3]

[1]Open-Source
[2]free for non commercial use, otherwise EUR 428,- for commercial license
[3]Open Dynamics Engine (www.ode.org)

3.4 Summary

Many robot simulators exists, each developed for another purpose but mostly all of them have the same abilities. They consist of a simulation-engine, some with integrated physics, and allow one or multiple clients to control and monitor one or multi robots in the virtual world.
The open-source simulators we introduced are all extendable, modifyable and they allow a better insight of the simulators behaviour. Furthermore the Simbad robot simulator includes two addtional standalone libraries that can be really helpful in developing evolutionary robotics.
Most commercial robot simulators provide enough flexibility to simulate complex robot control systems too, but due to their non-public sources it is difficult for the experimenters to adapt their programs for maximum efficiency.
To sum up, we can say that open-source robot simulators are more flexible than commercial robots simulators are. This is the main reason why they are preferred by hobbyists and academic researchers.

4 Problem Statement

The main aim of this thesis was to solve three problems regarding "Rossum's Playhouse" robot simulator. In this chapter we want to give you a short introduction what these problems were and why they were necessary to be solved. In detail these problems were:

- Synchronising Client-Server
- Implementing a Pollable Range-Sensor
- Implementing a Pivotable Range-Sensor

These modifications were needed to gain a higher simulation speed in our simulations, because we wanted to simulate a simple robot, equiped with a neural network, that should learn his way through a maze.

4.1 Synchronising Client-Server

Synchronisation between the client and server is necessary, when we have applications that are sensitive to timing issues or we want to run the simulation at an accelerated clock rate (in which the simulator time moves more quickly than the real-time). Another problem could be that our client's operations require a very large, time-consuming amount of processing so that the client can't keep up with the server. This was the problem we were facing with, because we wanted to simulate a neural network on our client. In our case it just happend that the client was still calculating when the server already sent the next sensor values. Therefore we needed a better synchronisation between our client and the server.

4.2 Implementing a Pollable Range-Sensor

A pollable range-sensor is useful when we want to know the current sensor-values at a particular time or don't want to receive permanently sensor-events. Moreover we wanted to have a range-sensor which only computes it's new sensor values on status-requests and not every time the robot moves. The idea was

to minimize the number of unnecessary computations on the client- and the server-side and with it to enable a faster simulation speed.

4.3 Implementing a Pivotable Range-Sensor

The idea of implementing a pivotable range-sensor was, that if we have a sensor we can rotate when we want and in any directions you want, it is not essential to place lots of sensors, each facing another direction, only to observe a larger area. With a simple rotation-request it is possible to turn the sensor towards the desired direction. As with the pollable range-sensor the main reason was the gain of simulation speed and the reduction of unnecessary computations on the client- and the server-side.

5 Implementation

In this chapter we present the solution to three problems that had been introduced in the last chapter. To recapitulate these problems were:

- Synchronising Client-Server
- Implementing a Pollable Range-Sensor
- Implementing a Pivotable Range-Sensor

5.1 Synchronising Client-Server

As we analysed the Rossum's Playhouse documentation and the sources we figured out that it already provides the possibility to synchronise the client with the server. The synchronisation is enabled/disabled by the option "Interlock" in the server configuration file. When enabled, it ensures a rigid synchronisation between the simulator clock and its client applications. This is achieved by suspending the simulation clock every time the server sends an event to the client. The clock is not re-started until the client process replies with an "interlock acknowledgement". This reply is not sent by the client until all event handlers related to the simulator-issued event have been invoked and have completed their operations. This ensures that all processing related to an event will be completed before the simulator is enabled to continue.
To enable the synchronisation we only have to add the following line to the server's property file:

```
interlockEnabled = true
```

5.2 Implementing a Pollable Range-Sensor

During our analysis we also come to know that pollable sensors are supported by the Rossum's Playhouse. On the client side polling is done by a sensor request to the server, which can be placed by calling the following method:

```
void sendSensorStatusRequest(RsBodySensor sensor)
```

This will trigger the server to send the current status for the queried sensor. The client will than receive an event but here comes the tricky part. How can we distinguish between events automatically generated by the server and our polled events? Because we are only interested in the polled events we don't have to take care how to distinguish between the two event types, we only have to find a way to disable the automatically generated events. The first and easiest part is to set the sensor resolution to 0. This will reduce the generated events to those generated by the robots motion. To eliminate these events completely we needed to modify the sources. Our aim was not to alter the existing functionalities but to extend them to meet our requirements. Because a pollable sensor event is triggered by the client's sensor request, we also came up with the idea to limit sensor value computations to those when the server has been requested to send the client the actual sensor status.
Now before we go into the details, here is a short overview about all required modifications in the Rossum's Playhouse:

- An interface for only pollable sensors
- Extend RsBodySensor and modify RsBodyRangeSensor
- Modify SimSensorRequestHandler
- Provide the client the ability to send a request to enable the sensor's "polling only"-mode
- Provide the server the ability to handle the client's request to enable the "polling only"-mode sensors

5.2.1 Interface RsIPollableSensor

The *RsIPollableSensor*-interface is intended to be implemented by sensors that can be switched into "polling-only"-mode. It defines the following three methods:

- `public boolean isPollable();` Returns the current value of *pollSensor*.
- `public void enable_SensorPolling(boolean enable);` Sets the new *pollSensor*-value.
- `public void compute_Sensor();` Calling this method sets the *computeSensor*-value true.

5.2.2 RsBodySensor and RsBodyRangeSensor

The class RsBodySensor has been extended by the following variables and methods:

- `boolean pollSensor = false;`
  Identifies if the sensor is in "normal" or "polling only"-mode.
- `boolean computeSensor = false;`
  When the sensor is in "polling only"-mode, no sensor values are computed until *computeSensor* is set true.
- `public boolean isPollable()`
- `public void enable_SensorPolling(boolean enable)`
- `public void compute_Sensor()`

Altough these variables and methods are visible to every sensor class, they are not used until they implement the *RsIPollableSensor*-interface. We implemented the interface only in the *RsBodyRangeSensor* class becausethis was the only sensor that was required to be pollable. The implementation is not complicated and involves the modification of one method only:

```
public boolean computeAndSetState(...)
```

As we mentioned before, the basic idea is that the sensor-values are not computed until a sensor request was placed. Therefore we added the following condition to the beginning of the method:

```
if (pollSensor && !computeSensor)
{
return false;
}
computeSensor = false;
```

The condition is only fulfilled when polling is enabled and the *computeSensor* has been set true by an incoming sensor-status-request (see the Server-Side Modifications).

5.2.3 Client-Side Modifications

In order to be able to enable the polling mode of a sensor, it was necessary to extend the RsClient- and the RsRequest-class.

- **RsClient**
  We added a new method to the class, which allows the enabling/disabling of the sensor's polling-mode.
  `void sendEnablePollSensorRequest(RsBodySensor sensor, boolean pollSensor)`
- **RsRequest**
  We added a new request-identification code to be able to identify the polling-enabling-request.
  `int REQ_ENABLE_POLL_SENSOR = 14;`

5.2.4 Server-Side Modifications

On the server-side request are first stored in a class, which is than processed by the request handler. Therefor a new class for storing the request `RsEnablePollSensorRequest` and a new request-handler `SimEnablePollingSensorRequestHandler` have been added to handle the `REQ_ENABLE_POLL_SENSOR`-request. When we take a closer look at both classes we can see that `RsEnablePollSensorRequest` is derived from `RsRequest` and that `SimEnablePollingSensorRequestHandler` is derived from another newly created interface `RsEnablePollSensorRequestHandler`. As we mentioned before a request for enabling/disabling the polling-mode will be only further processed if the corresponding sensor-class implements the `RsIPollableSensor`, this condition is checked by the request-handler class in the first place.
Also additional modifications where necessary in `RsConnection`:

- A new variable for the request-handler:
  `RsEnablePollSensorRequestHandler enablePollingSensorRequestHandler;`
- A new method to set the request-handler:
  `void setEnablePollingSensorRequestHandler(RsEnablePollSensorRequestHandler handler)`

5.3 Implementing a Pivotable Range-Sensor

In the case of the pivotable range-sensors no clue was found that it is already included in the Rossum's Playhouse. Hence we had to implement it completely by ourselves. Again modifications on the server- and client-side were unavoidable, but the solution is simple.
Now before we go into the details, here is a short overview about what these modifications are:

- An interface for pivotable sensors.
- Extend the existing "RsBodyRangeSensor", to be able to set the *sight-angle* every time we want.
- Extension of the RsProtocol, RsProtocolBodyDecoder and RsProtocol-BodyEncoder.
- Provide the client the ability to send a "Rotation"-request to the server.
- Provide the Server the ability to handle the client's "Rotation"-request.

5.3.1 Interface RsIPivotableSensor

The `RsIPivotableSensor` interface defines the method `public abstract void set_Angle(double _sightAngle)` for all pivotable sensor. The method allows to set the angle of the sensor at any time. Hence it is required that pivotable sensors implement this interface.

5.3.2 Class RsBodyPivotedRangeSensor

Because a pivotable range sensor is a sub-type of the normal range-sensor, we derived a new class `RsBodyPivotedRangeSensor` from `RsBodyRangeSensor`, that also implements the newly created `RsIPivotableSensor`-interface.

5.3.3 RsProtocol, RsProtocolBodyDecoder, RsProtocolBodyEncoder

Because we added a new type of sensor, we have to adapt the `RsProtocol-`, the `RsProtocolBodyDecoder-` and the `RsProtocolBodyEncoder-`class. Without these modifications the client won't be able to add a pivotable-range-sensor to the robot's body.

- **RsProtocol**
  A new body-part identification code was added to identify the sensor while encoding/decoding the body-part specification:
  `static final int BODY_PIVOTED_RANGE_SENSOR = 10;`
- **RsProtocolBodyDecoder, RsProtocolBodyEncoder**
  In both classes a new section has been added to the send/receive method. It is nearly identical to the `BODY_RANGE_SENSOR` section except that it the body-part identification code for the new `RsBodyPivotedRangeSensor` is used.

5.3.4 Client-Side Modifications

In order to be able to send rotation-requests of a sensor to the server, it was necessary to extend the RsClient- and the RsRequest-class.

- **RsClient**
  We added a new method to the class, which allows the enabling/disabling of the sensor's polling-mode.
  `sendRotateSensorRequest(RsBodySensor sensor, double angle)`

- **RsRequest**
  We added a new request-identification code to be able to identify the rotation-request.
  `int REQ_ROTATE_SENSOR = 15;`

5.3.5 Server-Side Modifications

As described in pollable-sensor's server-side modification, we needed to add a new class `RsRotateSensorRequest` to store the rotation-request and a new request-handler `SimRotateSensorRequestHandler`. As before the request-handler `SimRotateSensorRequestHandler` is implementing the newly created interface *RsRotateSensorRequestHandler*. To be sure to rotate only pivotable-sensors the handler first checks if the sensor is a pivotable-sensor anyway.
Again additional modifications where necessary in `RsConnection`:

- A new variable for the request-handler:
  `RsRotateSensorRequestHandler rotateSensorRequestHandler;`
- A new method to set the request-handler:
  `void setRotateSensorRequestHandler(RsRotateSensorRequestHandler handler)`

6 Evaluation

In this chapter we want to show you a performance comparison between the original Rossum's Playhouse sources and our modified version. For the comparison we used a simple robot equiped with a neural network that should find his way through a maze. The settings for the neural network are:

Population Size	20
Generations	200
Input Nodes	3
Output Nodes	2

The simulations were all running on the following machine:

CPU	AMD64 X2 6400+
Main Memory	2GB
OS	Ubuntu 7.10
Java	1.6

The original and the modified Rossum's Playhouse were tested at a simulation speed of 50 and 500 with 3 passes each to get an avarage result. In the next table we present you our results:

Sources	**Execution Time at 50x Sim.-Speed**	**Execution Time at 500x Sim.-Speed**
Original Rossum's Playhouse	failed[1]	12 minutes
Modified Rossum's Playhouse	27 minutes	9 minutes

[1]the simulation has failed all passes

Our modified version was 3 minutes faster than the original one at 500x simulation speed. In contrast to the original version the modified simulator was able to finish the simulation at 50x simulation speed. The original version always crashed after the 90th generation. This shows that our modified Rossum's Playhouse performs better than the original when running our neural network simulation on it.

7 Conclusion

Before we came to the analysis of some of the available robot simulation programs, we first gave you a short overview about robotics and simulation. The field of robotics involves a wide range of disciplines and going into to much details would have gone beyond the scope of this thesis.
Robot simulators are not only indented to simulate the robot's behaviour but also to monitor internal values and protect expensive hardware from being damaged or even being destroyed.
We introduced some of the most used robot simulators and came to the conclusion that it is better to use an open-source simulator like Rossum's Playhouse or Simbad robot simulator, because they are in no way inferior to commercial robot simulators in their functionalities and furthermore provide flexibility, extendability and the possibility to understand the simulator's functionality.
We also described in detail how we extended the Rossum's Playhouse by a pollable range-sensor and a pivotable range-sensor. The same way it is possible to add additional (yet unsupported) sensors or modify existing ones to satisfy requirements. Finally we compared our modified Rossum's Playhouse robot simulator to the original one and saw that it performs better than the orignal one.