Design of a Simulation and Visualization Tool for the Autonomous Football Table

R.P.W. Appeldoorn
r.p.w.appeldoorn@gmail.com

Supervisor: R.J.M. Janssen

Department of Mechanical Engineering
Section Control Systems Technology
Technical University of Eindhoven

September 6, 2011

Summary

In recent years the research on robotics and Artificial Intelligence (AI) has been increasing. It includes the development of robotic learning algorithms which involves a lot of testing. Developing these algorithms can be hard without knowing what information the robot is receiving from its sensors. Therefore, visualizing this sensor-data will make it easier for the developer to understand what the exact issues are.

Together with this visualization, it is useful to have a simulator on which these algorithms can easily be tested which saves a lot of costs and time. This thesis will focus on the development of a simulation and visualization tool for an autonomous football table. The tool consist of three main elements: the simulator, a middleware and a graphical user interface.

The creation of the visualization is done with use of an Open Source 3D Graphics Engine (OGRE). With use of this engine, a rendering window for 3D-models can be created. Together with the QT-framework, which is used to develop applications on multiple platforms, a tool is created to monitor and visualize the sensor-data of the robot. The link between this tool and the actual control is created with use of the Yet Another Robot Platform (Yarp) middleware. This middleware provides a set of tools for easy interaction between multiple programs in different languages.

The creation of the simulation is done with use of the Multi Open Robots Simulator Engine (MORSE). This software provides interaction between a client program and a simulator via empty-sockets or several middlewares. The simulator is based on the game-environment in Blender (3D-modeling software) which includes realistic 3D-dynamics. Python-scripts are linked to this simulator and act as sensors and actuators that communicate with the client. In our case, the YARP middleware is used to exchange the reference signals and the sensor data between the actual control and the simulator. This data can be monitored with use of the visualization part in the tool.

The final program provides a tool to concentrate on problems such as real-time machine learning and development and tuning of higher level gameplay strategies.

Acronyms

AI Artificial Intelligence

IDL Interface Defination Language

OS Operating System

MORSE Multi Open Robots Simulator Engine

Yarp Yet Another Robot Platform

ROS Robot Operating System

GUI Graphical User Interface

ODE Open Dynamics Engine

OGRE Open Source 3D Graphics Engine

CAD Computer Aided Design

Contents

1 Introduction

More often, simulators are used to easily test robots with use of a simulated environment. Tests can be performed without depending physically on the actual machine. This saves the user a lot of time and costs. Besides a simulation, it is very useful to know what the robot is seeing, thinking and doing during a workout. Developing robot software can be very difficult without exactly knowing what the robot thinks that is going on in the world. Therefore, besides a simulator, it is really useful to visualize the sensor data of the specific robot so that the user obtains the same information as the robot does.

It is helpful to develop a tool for these two elements: simulation and visualization. This tool needs data exchange between the real robot, the control and the different parts of the tool. This can be done with use of a robotic middleware. This middleware has to provide a set of software that makes communication between these subsystems possible. After implementation of such a tool, the project structure will expand as shown in figure 1. The final tool has to form a simple base for studying and developing situated Artificial Intelligence (AI) and machine learning algorithms.

In this report, the development of a simulation and visualization tool for the autonomous football table will be described. Firstly, several middlewares and simulators will be discussed. Secondly, the implementation of this simulator and middleware will be described. At last, the visualization tool will be evaluated and the total results of this project will be shown.

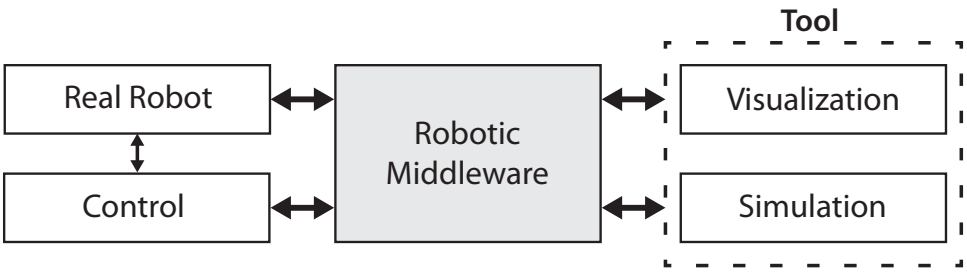


Figure 1: Expanded project structure

2 Simulator/Middleware comparison

2.1 Simulators

Simulators mostly consist of three primary elements: dynamics engine, sensors/actuators and a rendering system for visualization. In this subsection, the two commonly used 3D-simulators will be discussed and compared.

2.1.1 Gazebo

Gazebo is an open source 3D robot simulator which can easily interact with the Player middleware. It is capable of simulating a population of robots, sensors and objects. Gazebo provides a large number of sensors and actuators for the user to work with. It generates realistic sensor feedback to external programs and it is capable of simulating rigid-body physics. [1].

Gazebo makes use of ODE as dynamics engine and OGRE as 3D rendering system. These two components together with the sensors and controllers, form the main part of Gazebo.

A gazebo simulation consists of the following parts:

- **World-file:** This is a XML-file that can be parsed by Gazebo. It includes specific model files which together form the simulation scene. It also contains the specific ODE-settings for that specific simulation.
- **Model-files:** Files with information of the bodies, geometries, joints, sensors and actuators of the models written in XML-language. Bodies can consist of multiple geometries of simple shapes and trimeshes. These bodies can be connected with each other with use of joints which form the total model. Joints can be manipulated or analyzed with use of sensors and actuators which communicate with a C++ program.

Gazebo can communicate with the middlewares ROS and Player. It is also possible to get direct communication with the simulator with use of the libgazeboC++ library.

2.1.2 MORSE

MORSE is a versatile simulator for generic mobile robots simulations. It is enabling realistic and dynamic environments with use of Blender as 3D rendering system and Bullet for the physics simulation. Besides that, it is compatible with various robot middlewares [2].

Controllers, sensors and other objects in MORSE are entirely scriptable in Python. This makes it easy to generate new variations of these objects. MORSE uses the GameLogic module implemented in Blender which is a module for game creation. It provides a game-engine with Python script interactivity and realistic physics. With use of the MORSE-addition to this module, it is possible to communicate with other clients using an empty-socket or various middlewares.

A MORSE simulation consists of the following parts:

- **Objects:** Blender objects with various geometries that can contain material and physical properties. These objects can be created with use of Blender but it is also possible to import models from other Computer Aided Design (CAD) programs.

- **Robots:** Objects that are linked to a Python script that is called every time step. The robot-objects can contain other objects with optional sensors and/or actuators.
- **Sensors:** Child of a Blender robot that is linked to a Python script. This script can gain information of its parent in the simulated environment and save it to local variables.
- **Actuators:** Child of a Blender robot that is linked to a Python script. It can manipulate objects in the simulated environment with use of input via the local variables stored in the instance of the Python-class.
- **Middleware-communicator:** These elements create a connection between the MORSE-simulator and the specified middleware. It has the ability to manipulate and read the local variables stored in the instances of Python-classes of the sensors, actuators and robots.

In appendix A.1, the MORSE data flow between the components mentioned above is shown.

The total simulation scene can be build with use of the GUI that comes with Blender. This provides a clear overview of all the elements that MORSE has to offer. The simulation can be controlled with use of Python scripts, middlewares and client programs..

2.1.3 Comparison

In order to communicate with the simulator, a client program has to be implemented. This client has to be written in C or C++. This is because these two languages can be implemented in Matlab Simulink which is used to control the football table. The implementation of these two languages can be done with use of S-functions. Since Gazebo is entirely programmable with use of C and C++, no external middleware is needed for this simulator. This is an advantage of the Gazebo simulator.

When looking at the simulator itself, MORSE is much more user friendly than Gazebo. It uses the GUI of Blender in which all simulation parameters can be set. In Gazebo, the total simulation scene has to be written in XML-language. Therefore it is not clear what the exact options are for each element. Since the documentation of Gazebo, at the moment of writing, is very outdated, it is hard to start with this simulation program.

Another advantage of MORSE is the opportunity to import various CAD-models of different formats. When a model of your robot is already available, it is much easier to implement this model into MORSE then implementing it into Gazebo. In Gazebo, the total model has to me remade or part by part imported which involves a lot of XML-writing.

Overall, both simulators can be used to simulate robots, sensors, actuators and physics in a three dimensional world. The communication between the client and the simulator would be easier using Gazebo. But this problem can be solved using the middlewares supported in MORSE. Our choice will be MORSE since it is much easier to implement with existing models and creating the simulation can almost totally be done using the GUI of Blender.

2.2 Middleware

Robotics middleware is the link that connects the user software with the robot (simulated) hardware. Often, only communication between components is considered to be middleware. But middleware varies from just communication to application-independent code that helps the system developers in the composition of subsystems into larger systems.

In this section, three robotics middlewares will be discussed and compared.

2.2.1 Yarp

Yarp is a set of software which acts as a communication plumbing for the robotics software. It is a set of libraries, protocols and tools to keep modules and devices cleanly decoupled [3]. Yarp is reluctant middleware which features can be used to improve the current software running on the different subsystems.

Yarp consists of three main components:

libYARP_OS The main task of this component is interfacing with the different operating systems by streaming data across many machines. It is OS neutral so it can be used by different platforms. Almost the entire communication of YARP is written in C++.

libYARP_sig Performs common signal processing tasks in an open manner. Easily interfaced with common libraries as OpenCV.

libYARP_dev Interfacing with common devices such as cameras, motor control boards etc.

Yarp works well with the MORSE simulator. It gives full support for all sensors and actuators that are used in MORSE [6].

The most important component of Yarp in case of the football table is the libYARP_OS. This component provides the 'Yarp Server' that creates a way to easily communicate between the different components in the system.

2.2.2 ROS

The Robot Operating System (ROS) is a thin, multi-lingual, tool-based system designed for mobile manipulators [4]. To work without depending on other libraries, ROS is composed with reusable libraries. The ROS specification is at the messaging layer, this means that the libraries are outlayered. It uses a simple Interface Defination Language (IDL) to communicate between programs that make use of these libraries. These programs are called ROS nodes and can be written in multiple languages.

A ROS node is a process that performs computation. These nodes are organized into packages, stacks and ultimately applications. Packages can almost contain anything like message definitions, tools, nodes and libraries. Stacks provide a set of packages and applications are used for the execution of a robot program.

ROS being a robotic operating system allows a developer to easily integrate sensors and actuators with software procedures, assuming the device driver exists. It offers all the services required

for an operating system, which includes passing of message between two systems, controlling low-level devices, and hardware abstraction. In order to develop, build and run software across a network it also provides the required libraries and tools.

The ROS middleware supports both simulators (MORSE and Gazebo). It provides full support for the Gazebo simulator, but the support for MORSE is still in development [7].

2.2.3 Player

Player is a network server for robot control. It maintains the robot actions as a network based server that is running on your robot. It gives a clear and simple interface to robot sensors and actuators via a TCP-IP connection. A client application communicates through a TCP socket to the player and performs actions like reading data from the sensors, posting commands to actuators, and configuring devices [5].

The Player software is language and platform independent. It supports multiple programming languages such as C, C++, Ruby and Python [5].

Player interacts with robots devices using specific device drivers. It provides a standard device interface to its clients so that it provides a more generic control method. Custom drivers can be written for robots and devices so that they can interact with the player server as different modules.

Player can interact with the Gazebo simulator. It contains drivers of controllers/sensors and robots used in Gazebo.

2.2.4 Conclusion

Yarp provides interface and communication abstractions that creates the link between several subsystems. It is a toolbox for communication and some other aspects for signal analyses and device communication. Yarp is transport-neutral so it is easy to link multiple platforms together with use of Yarp Server.

ROS is primarily designed for complex mobile manipulation platforms. ROS comes in a distributed computing environment. Unlike Player, that offers hardware drivers, ROS offers implementation of algorithms. Though ROS is more powerful and flexible its complexity makes a little difficult to use.

Player is the most widely used robot interface amongst the three. The Player robot server architecture provides a set of modules, known as drivers, that can communicate with different devices or programs. Player itself provides a generic interface to the client program so that the control happens in a generic way. This can be done via a TCP-IP connection.

In our situation, the main task of the middleware is the communication between the simulator (Python) and the client program (C++). Since our choice for the simulator is MORSE, the supported middlewares that are left are Yarp and ROS. Because most of the elements in MORSE are not supported by ROS yet, Yarp is the best middleware to use in this project. With use of the Yarp Server, a bridge can be established between the simulator, our client program and the real robot.

2.3 Evaluation MORSE with Yarp

With use of Yarp as middleware and MORSE as simulator, it is possible to control the simulator with a client program written in C++. The client can easily be implemented in the current control model of the football table. With use of libYARP_OS, data can be sent and received via the Yarp Server to the controllers and sensors on the simulator side. The libYarp_sig library provides tools to easily process the images of the simulated camera to a readable format for the existing vision control. A schematic overview is given in figure 2.

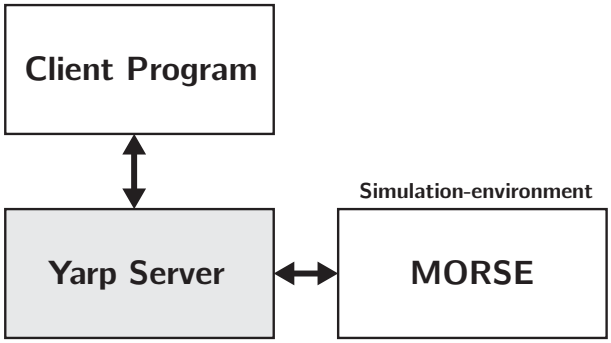


Figure 2: Overview Client/Simulator communication via Yarp Server

3 Implementation Simulator with MORSE and Yarp

In this section, the set-up of a simulation in MORSE with use of Yarp is described. Firstly, the creation of the simulation will be discussed. Secondly, the communication between the client and the simulator will be explained. A 'How-To' of installing MORSE 0.3 and Yarp on Ubuntu 10.04 is included in appendix A.2.

3.1 Robots

In this subsection, the creation of the football table robot is discussed. At the end of this subsection, the model is assigned as a robot and it will be possible to add controllers and actuators for client-interaction with use of middlewares.

3.1.1 Creation of the geometries

Since MORSE uses Blender, it is easy to import models from all different file-formats. CAD-models from external programs can be exported to .STL-files, which are imported in Blender. In case of the football table, the only complex object of the football table is the 'Puppet'. The other objects can simply be made with use of standard geometries (box, sphere, cylinder). The total model of the football table is shown in figure 3. This model does not contain any physics properties, controllers or sensors.

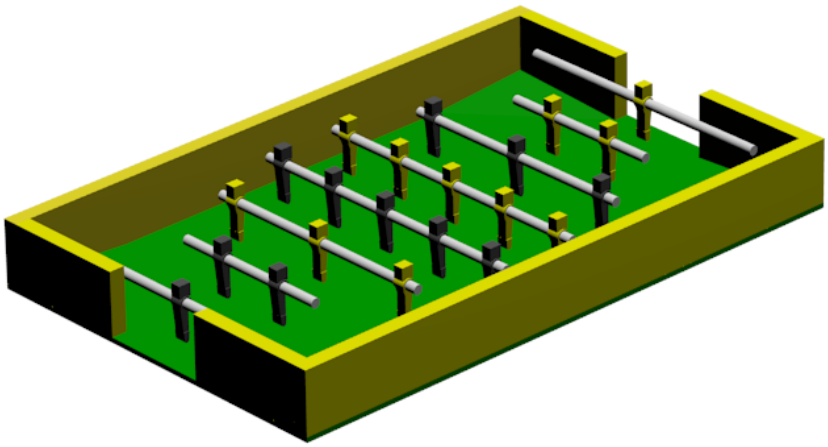


Figure 3: Blender model of the football table

3.1.2 Physics properties, Constraints and Bounding boxes

To take the different objects in the model of the football table into account of the physics simulation, some properties need to be set up. For every object in the model, a physics type can be set. In our case, the objects that form the table are chosen to be static. This is because the movement of the objects can be neglected.
The other objects (Rods and Puppets) can be interpreted as solid bodies with no deformation. Therefore, these objects will be set as 'Rigid body'. Also the mass, damping and other properties of all objects have to be specified to create realistic simulation results. It is necessary to tune these

values so that the behaviour of these objects match the behaviour in the reality.
On the physics properties panel of the Blender interface, it is possible to put restrictions on different objects in the scene. In our case, the rods have two degrees of freedom: x and φ. These constraints can be set to their local axis.
The last task in the physics section is applying a bounding-box to the objects. This bounding-box is used to detect collision between objects. The model mostly consist of standard geometries for which the bounding-boxes are equally simple. The puppet-objects need more complex bounding boxes to gain a realistic simulation. A triangle mesh is used to cover the complex geometry of the puppet. This bounding-box costs much more computational effort because the number of different contact planes increases which requires more computation.

3.1.3 Make it a robot

The model is now correctly set up to work with the physics engine but an external output is not possible yet. This is the part where MORSE separates itself from the Blender GameLogic Module. The model has to be recognized as a robot to be scriptable with use of Python via a specific middleware. Therefore, a *Robot_Tag* and some other properties have to be added to the model (Appendix A.3). These properties link the robot model to a specific Python class with an initialization and default action function that is called every time step (Appendix A.4). This class can be used to store or modify values of the concerning robot.

3.2 Controllers

Now that the model is recognized as a robot, controllers can be added to the objects of the model. In our case, every rod needs an actuator for $x-translation$ and $\varphi-rotation$. This can be done by adding an empty-object as child of the rod object. To make this object an actuator, a Python class has to be linked to this object (Appendix A.3) and, instead of a *Robot_Tag*, a *Component_Tag* is applied. This class contains two functions: initialize function and a default-action function which is called every time step (Appendix A.5). The controller can apply actions to its parent to manipulate its position or orientation. A schematic overview of the controller is given in figure 4.

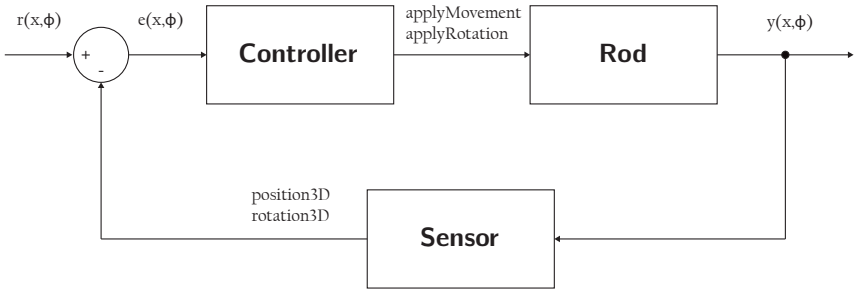


Figure 4: Rod position controller

This controller functions like a regular proportional-controller (P-controller). But instead of applying a force, a velocity is applied to the parent object with use of the functions *applyMovement* and *applyRotation*.

3.3 Sensors

To construct a good reference signal, the positions of the ball and the rods are needed. Therefore, sensors are needed that return these values to the client. Because it is a simulation, the values do not have to be 'measured', but can simply be gained via a query to the simulator. This gives the exact position without any disturbance. Whereas in the real world a camera is used to measure the positions of the ball in the field and encoders to measure the rod-orientations. These sensors also have to be linked to a Python-script (Appendix A.3) and get a $Component\_Tag$. In case of the football table, two different sensors are needed to provide data to the client:

- **Position-Sensor:** Provides the x and φ values of the rods.
- **Ball-Sensor:** Provides the x,y and z values of the ball.

The third and optional sensor is the camera sensor that simulates the real camera that is placed above the table. It provides camera data to the client at a specific frequency. This data can be analysed with the actual actual vision algorithms for tracking the ball position. Comparing the values of the Ball-Sensor and the calculated ball-positions of the camera data, gives the exact error of the analysed ball position.

3.4 Yarp and Yarp server

The Yarp middleware is used for the interaction between the simulator and the client. This can be done with use of the Yarp Server. The simulator and the client both act as clients to the Yarp Server.
To create the client on the simulator side, a Yarp middleware object has to be added to the scene. This object has to be linked to the objects with the 'Component_Tag'. This has to be done in the file '$component\_config.py$' which, for our project, is given in appendix **??**. When the simulation starts, local ports on the simulation side are created to exchange data with the external clients. The addresses of these ports always have the following form:

$$/ors/robots/(robot\_name)/(object\_name)/(in/out)$$

For every connection of the external clients to the simulator, a port has to be created with an internal and external address. When connecting a port, Yarp Server will establish a link between these two addresses on the different subsystems. In case of the football table, four ports are needed:

- Port for sending reference signals of the rods to the simulator
- Port for receiving the positions and rotations of the rods
- Port for receiving the position of the ball
- Port for receiving the camera data

The code for initializing these ports is given in appendix A.7.

Now that the ports are established, interaction can take place via this port with use of specific communication messages. These messages of Yarp have specific forms and are called 'bottles'. Bottles are tulips; objects that can contain multiple variable types.

These bottles can be sent and received as follows:

Sending bottles via Yarp Server The procedure of sending a bottle in Yarp consist of three steps: The first step is the initialization of the bottle. The bottle is created as an instance of the bottle class of Yarp. The second step is adding values to the specific bottle. Now that the bottle is filled with data, it can be sent, this is the last step. The bottle is converted to a simplified format and sent via the Yarp Server over the specific port. The C++ code of sending a bottle over the reference-signal-port is given in Appendix A.8.

Receiving bottles via Yarp Server Reading the sent bottles is done with use of the following routine: Check if a bottle is available, get the bottle from the specific port, read out the values of the bottle with use of indexes. The code for receiving the sensor values is given in Appendix A.9.

3.4.1 Communication overview

Figure 5 shows an overview of the data flow via the ports when interacting with the simulator:

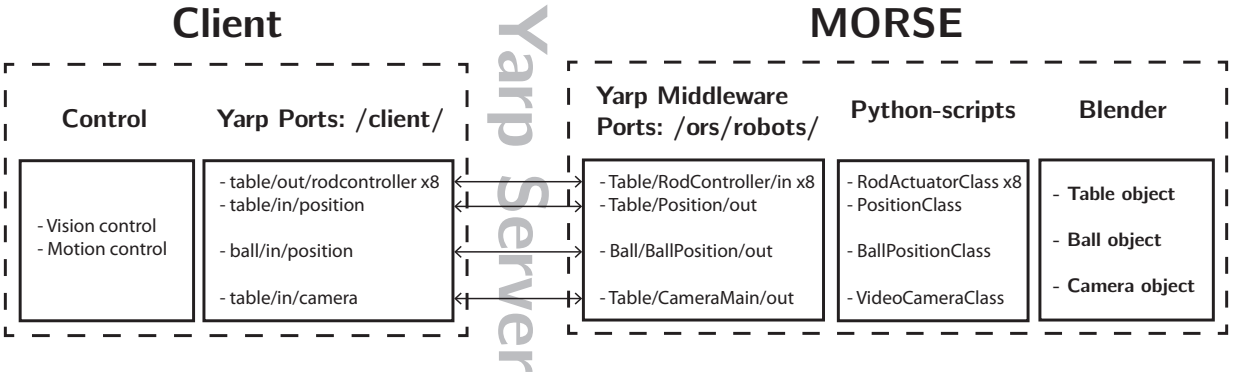


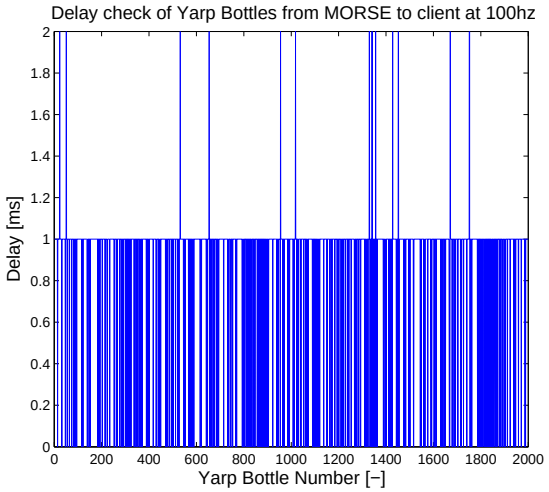
Figure 5: Ports overview

4 Analyses of Yarp with MORSE

To gain insight in the performance of the different parts in our system, the delay of every separate element has to be analysed. The results of this analysis is are shown in this section.

4.1 Yarp communication delay

The Yarp Server delay is checked with use of two simple C++ clients that interact with each other using the Yarp Server. The current time in milliseconds is sent by the *sender* to the *receiver* on the other side. The receiver compares the 'send-time' with the current time which forms the delay. The two C++-codes are given in appendix A.10. With this test, the bottle-loss and the bottle-delay can be analysed for different send-frequencies. Figure 6 shows the communication-delay for different frequencies.



Frequency:	Average Delay:	Bottle-loss:
100 hertz	0.1161 ms	0%
1000 hertz	0.1163 ms	2.1%
10000 hertz	0.1596 ms	40.1%

Figure 6: Delay and bottle-loss analysed for different frequencies.

As can be seen, the communication with a frequency of 100 hertz has the best performance. When increasing the frequency, bottle-loss will occur; the bottle-delay will stay more or less stable. Since the current vision-loop runs on a frequency of 100 hertz, it is logical to also apply this frequency to the Yarp-communication. This gives a minimum loss of bottles and a minimum average delay of 0.1161 ms.

4.2 MORSE Yarp Interaction

MORSE consists of Blender with the GameLogic module and a middleware to communicate with external programs. In our case, MORSE interacts via a Yarp-middleware-communicator which is a client to the Yarp Server. The delay between the control-client and the middleware-client is analysed in the previous subsection.

The interaction between the client program and MORSE is analysed using the same method. Instead of using a C++-program as a sender, a sensor in MORSE is used (Appendix A.11). This sensor transmits the current time to the receiver used in the previous subsection. It sends 1000 bottles of the current time which are compared again at the receiver side. The results of this test is shown in figure 7.

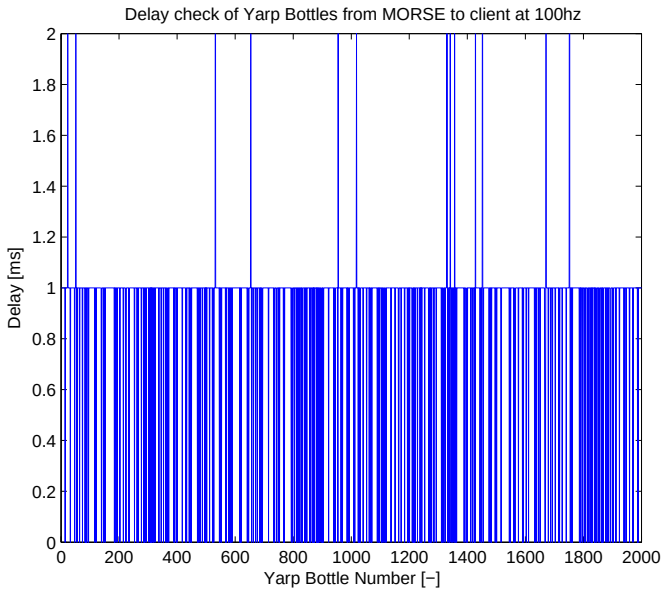


Figure 7: Analysis of the MORSE-client interaction - Average delay: 0.8459 ms; Bottle-loss 0%

This test is performed at a simulation frequency of 100 hertz. This frequency represents the update-rate of every script that is linked to the GameLogic module of Blender. Out of this test, it can be concluded that the total delay of the communication between the client and the simulator has an average of 0.8459 ms.

This delay is a representation of the communication under perfect circumstances. When running a real simulation, the update-frequency of the simulator will not be this constant due to computational limitations. Therefore the delay will increase depending on the performance of the machine that runs the simulation.

5 Implementation of the Graphical User Interface

5.1 Intention of the GUI

Because of the fact that the whole system consists of little subsystems, it is helpful to develop a tool that covers all subsystems. This tool has to take care of starting and connecting every subsystem with each other. With use of a small pair of buttons and a visualization, it has to provide an application that is much more user friendly and easier to control. Figure 8 shows a schematic overview of the total project set-up. The GUI takes care of setting up, connecting and monitoring the multiple subsystems.

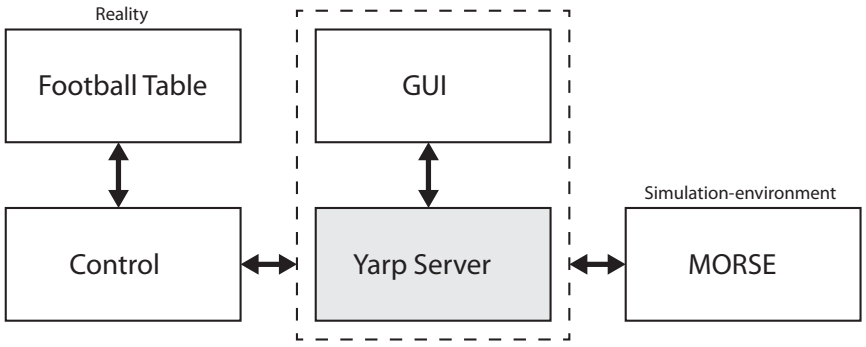


Figure 8: Total overview

5.2 Clients

As can be seen in figure 8, the Yarp Server is integrated in the GUI. This can be done by initializing a Yarp Server process when the GUI is initialized. With use of this method, the output of the Yarp Server can be displayed in the GUI and no external window is needed.
Now that the Yarp Server is set up, the clients can connect to the server. In case of the football table, three client connections are needed:

- **Control client:** Sending reference signals, receiving sensor information (Simulation), sending sensor information (Reality)
- **GUI client:** Receiving sensor information and reference signals for visualization
- **MORSE client:** Receiving reference signals and sending sensor information

Because of this set-up, the same control for the actual football table can be applied on the simulator.

5.3 Visualization

A visualization is implemented in the GUI so that the user can see what information the robot obtains during a work-out. This part of the GUI visualizes the sensor data that comes from the GUI-client with use of OGRE (3D-rendering system). Updating this visualization and receiving data is done at a frequency of 50 hertz. This provides a smooth playback of the sensor data of the (simulated) robot to the user.

5.4 Result

A graphical representation of the GUI is given in figure 9:

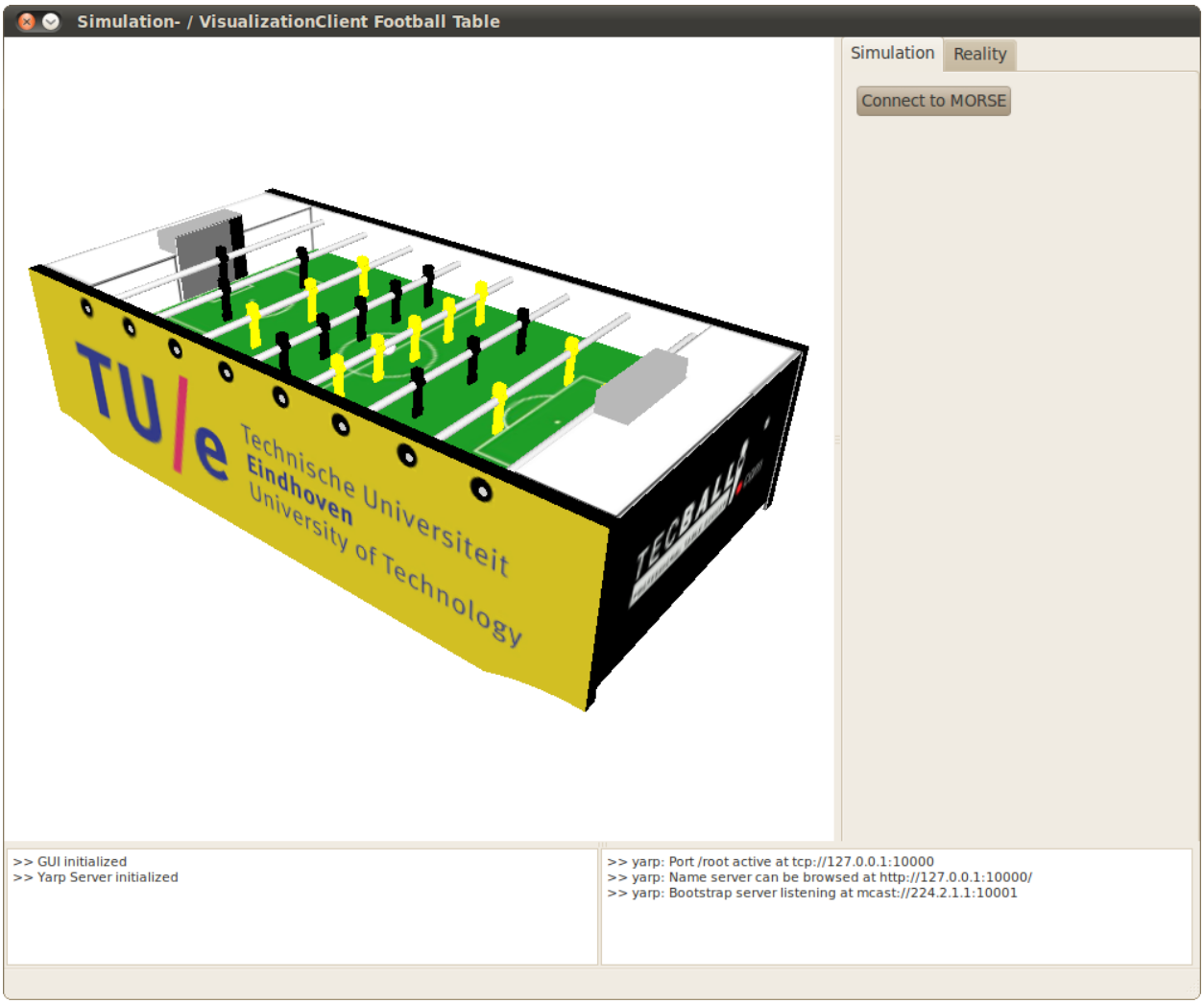


Figure 9: Image of the Graphical User Interface

As can be seen in figure 9, the GUI allows the user to switch between two options:

- Simulation
- Visualization of the reality

6 Conclusion and further remarks

6.1 Project results

The main output of the project is the development of a simulator for the autonomous football table. Besides this, the project structure is expanded with use of a central Yarp Server for communication between the subsystems. On top of that, a GUI is developed to make control of the system much easier with use of one central application.

With use of this simulator, tests can be performed at a faster pace. This makes it much easier to construct and test learning-scripts for robots. The simulator works in a perfect environment with perfect sensors and controllers. This is a disadvantage of these simulations because the reality is much more complex that what the simulator offers. Therefore, some future work has to be done and some safety factors has to be taken into account.

6.2 Future work

- **Optimization of the simulator:** For the use of force and torque on different elements in the simulation, the dynamics of the elements have to be realistic. Tests have to be done to optimize the dynamic parameters so that the simulation behaves as close to the reality as possible. Parameters that have to be tuned: inertia, elasticity, damping, friction etc.
  To create more realistic sensor feedback, it is also possible to apply a modifier to the outgoing data from the sensor. This adds a Gaussian noise to this signal so that the values are more realistic.

- **Improvement of the controller:** The current controller works with $applyMovement$ and $applyRotation$. This means that it starts moving with a constant velocity without a change of acceleration. To make the controller more realistic, a controller can be made that applies force to the different rods with use of the $applyForce$ and $applyTorque$ methods. By doing so, the controller used for the actuation of the real-rods can be used for both: Reality and Simulation. This is because the voltages that are sent to the real actuators are directly proportional to the forces. To gain this result, the ethercat used in the reality has to be simulated as well. In this way, the plant can simply be substituted by the simulator when running a simulation.

- **Implementation of vision-algorithms on camera-sensor:** In the current simulation, the position of the ball is captured with use of the $position3d$ function in Blender. To make this sensor output more realistic, modifiers can be added. But all this can also be done by analysing the camera-sensor just as it happens in the real world. Therefore, the vision control has to be applied on the sensor output-data. By doing so, the simulation becomes more realistic and the same occlusions will happen in the simulation as in the reality.

- **Improvement of movement-visualization:** The current GUI gives only a representation of the movement of the ball and the movement of the rods. It would be more interesting to also visualize the tracking paths of the vision algorithms to check whether they are right or wrong. This can be done by adding these visual effects to the Ogre Class. This provides a better overview for the users and can help in developing better algorithms for the control.

References

[1] http://playerstage.sourceforge.net/index.php?src=gazebo

[2] http://www.openrobots.org/morse/doc/stable/morse.html

[3]] http://eris.liralab.it/yarpdoc/what_is_yarp.html

[4] http://www.cs.stanford.edu/people/ang/papers/icraoss09-ROS.pdf

[5] http://playerstage.sourceforge.net/index.php?src=player

[6] http://www.openrobots.org/morse/doc/latest/user/supported_middlewares.html

[7] http://homepages.laas.fr/slemaign/publis/Echeverria2011.pdf

A Appendix

A.1 MORSE Data flow diagram

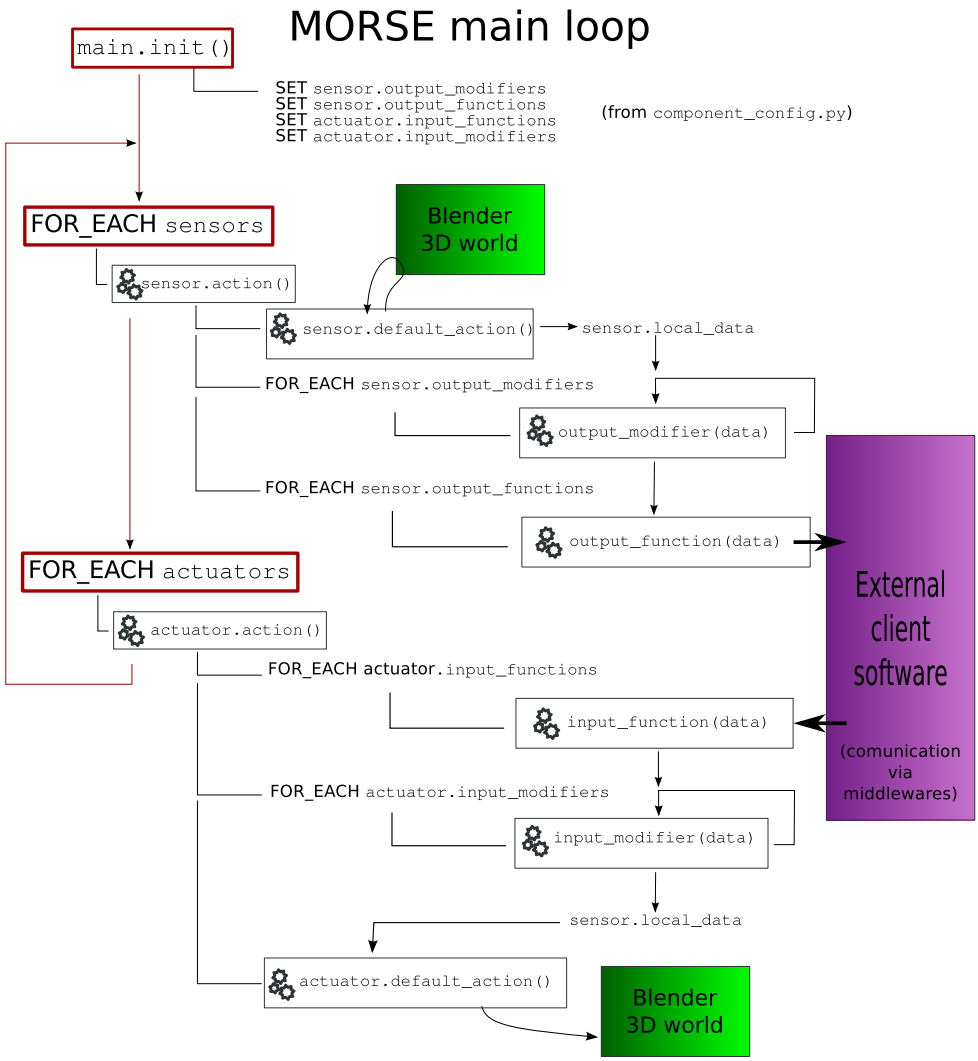


Figure 10: The data flow between components of the MORSE simulator [7]

A.2 How to install MORSE with Yarp

This installation guide describes the installation of MORSE 0.3 with Yarp support on Ubuntu 10.04 LTS.

Dependencies

- sudo apt-get install swig
- sudo apt-get install python3.1-dev

- sudo apt-get install libgtk2.0-dev
- sudo apt-get install subversion build-essential gettext \
  libxi-dev libsndfile1-dev \
  libpng12-dev libfftw3-dev \
  libopenexr-dev libopenjpeg-dev \
  libopenal-dev libalut-dev libvorbis-dev \
  libglu1-mesa-dev libsdl1.2-dev libfreetype6-dev \
  libtiff4-dev libsamplerate0-dev libavdevice-dev \
  libavformat-dev libavutil-dev libavcodec-dev libjack-dev \
  libswscale-dev libx264-dev libmp3lame-dev

Installing Yarp

- sudo apt-get install cmake libace-dev subversion
- svn co https://yarp0.svn.sourceforge.net/svnroot/yarp0/trunk/yarp2 yarp2
- cd yarp2
- mkdir build
- cd build
- ccmake ..

% Make sure that you have set 'Build Python' enabled and link the right paths (Python3.1)

% For Yarpview, enable set_guis. Make sure you have got the GTK+ dev library. Press 'c' and 'q'.

- make
- sudo make install

Link SWIG

- cd ..
- cd examples/swig
- ccmake .

% Make sure you have Python 3.1 enabled and press 'c' and 'q'

- make
- sudo make install

Installing ACE

- ../scripts/fetch-ace.sh
- cmake ..
- make
- sudo make install

Installing Blender

- Blender-2.56-beta-source: http://download.Blender.org/release/Blender2.56beta/
- cd Blender2.56beta
- scons (compiles Blender in ../install and ../build)
- copy the Blender in the ../install to the location you want

Installing MORSE

- Morse-source: https://github.com/laas/morse/tarball/0.3
- mkdir build
- cd build
- cmake ../
- make
- sudo make install

Set environment variable

- sudo gedit /etc/environment
- add line: MORSE_Blender=/opt/Blender256b/Blender (path to Blender executable)
- restart your computer

Start-up MORSE and check if the Yarp-bindings are correctly set-up

- morse
- shift+f1
- add middleware yarp_empty (share/morse/components/middleware/yarp_empty.blend /object/yarp_empty)
- Press 'P' to run the simulation

A.3 MORSE settings for Python-script linking

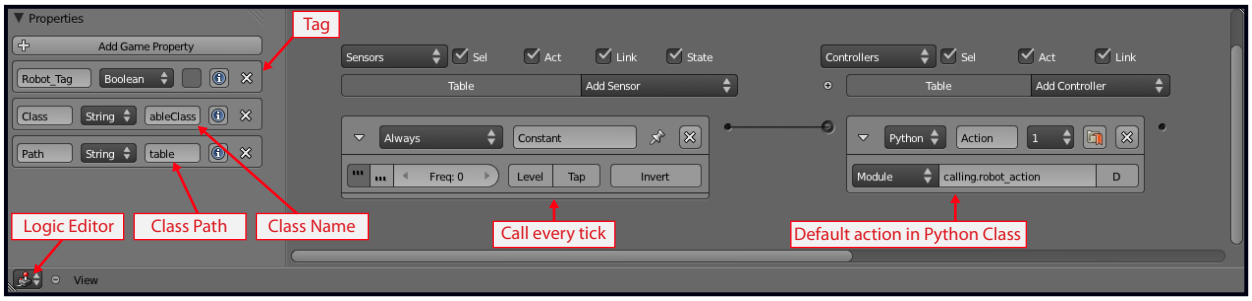


Figure 11: Adding a Python script to a Blender component

A.4 Robot Class of the Table

```
import GameLogic6
import morse.core.robot

class TableClass(morse.core.robot.MorseRobotClass):
    """ Class definition for the TABLE-ROBOT.
        Sub class of Morse_Object. """

    def __init__(self, obj, parent=None):
        """ Constructor method.
            Receives the reference to the Blender object.
            Optionally it gets the name of the object's parent,
            but that information is not currently used for a robot. """
        # Call the constructor of the parent class
        print ("######## ROBOT '%s' INITIALIZING ########" % obj.name)
        super(self.__class__,self).__init__(obj, parent)

        print ('######## ROBOT INITIALIZED ########')

    def default_action(self):
        """ Main function of this component. """
        pass
```

Listing 1: Robot Class of the Table (Python)

A.5 Rod Actuator Class

```
class RodActuatorClass(morse.core.actuator.MorseActuatorClass):
        """ Motion controller using linear and angular position

        This class will read linear and angular position (x, phi)
        as input from an external middleware, and then apply them
        to the parent robot.
        """

        def __init__(self, obj, parent=None):
                print ('######## x-phi CONTROLLER INITIALIZATION ########')
                # Call the constructor of the parent class
                super(self.__class__,self).__init__(obj, parent)

                self.local_data['x'] = 0.0
                self.local_data['phi'] = 0.0
```

```
16
17                  print ('######## CONTROLLER INITIALIZED ########')
18
19          def default_action(self):
20                  #Define some constants
21                  gain_velocity_x = 0.3
22                  tolerance_x = 0.001
23
24                  gain_velocity_phi = 0.01
25                  tolerance_phi = 0.001
26
27                  #Initialize the object
28                  parent = self.Blender_obj.parent
29
30                  #Calculate Phi
31                  phi_c = math.degrees(math.acos(parent.orientation[2][2]))
32                  if(math.asin(parent.orientation[2][1]) > 0):
33                          phi_c = 360-phi_c
34
35                  # Read out the position and rotation of the rod
36                  [x, phi] = [parent.position.x,phi_c]
37
38                  # Read out destionation position
39                  [dest_x, dest_phi] = [-self.local_data['x'],-self.local_data['phi']]
40
41                  # Set the rod to the certain x-position
42                  error_x = dest_x-x
43                  velocity_x = error_x*gain_velocity_x
44                  parent.applyMovement([velocity_x,0,0], True)
45
46                  # Set the rod to the certain phi-position
47                  error_phi = dest_phi-phi
48                  if(error_phi>180):
49                      error_phi = -(360-error_phi)
50                  elif(error_phi<-180):
51                      error_phi = (360+error_phi)
52                  velocity_phi = error_phi*gain_velocity_phi
53                  parent.applyRotation([velocity_phi,0,0], True)
54                  #print("phi:" + str(phi) + " error_phi:" + str(error_phi))
```

Listing 2: Rod Actuator Class (Python)

A.6 Linking Blender objects to middleware

```
1  """ Configuration of the middleware to be used by each component. """
2
3  # Every component is listed manually, with the type of middleware it should be using
4
5  # Middleware Components
6  component_mw = {
7          "RodController0": ["Yarp", "read_message"],
8          "RodController1": ["Yarp", "read_message"],
9          "RodController2": ["Yarp", "read_message"],
10         "RodController3": ["Yarp", "read_message"],
11         "RodController4": ["Yarp", "read_message"],
12         "RodController5": ["Yarp", "read_message"],
13         "RodController6": ["Yarp", "read_message"],
14         "RodController7": ["Yarp", "read_message"],
15         "Position": ["Yarp", "post_message"],
16         "BallPosition": ["Yarp", "post_message"],
17         "CameraMain": ["Yarp", "post_image_RGBA"],
18
19
20         }
21
22 # Components that will use a modifier
```

```
component_modifier = {}
```

Listing 3: component_config.py

A.7 Initialize Yarp Server and Ports

```
#include "morseclient.h"
#include <yarp/os/Network.h>
#include <yarp/os/BufferedPort.h>
#include <yarp/os/Bottle.h>
#include <yarp/os/Time.h>

using namespace std;
using namespace yarp::os;

BufferedPort<Bottle> PositionCommunicationPort;
BufferedPort<Bottle> BallPositionCommunicationPort;
BufferedPort<Bottle> RodCommunicationPort[8];

// Connect to the MORSE-Server
bool MorseClient::Connect() {
    QString table_robot_adress_prefix = "/ors/robots/Table/";
    QString table_local_adress_prefix = "/table_client/table";

    QString ball_robot_adress_prefix = "/ors/robots/Ball/";
    QString ball_local_adress_prefix = "/table_client/ball";

    QString PositionOutputAdress = table_robot_adress_prefix + "Position/out";
    QString BallPositionOutputAdress = ball_robot_adress_prefix + "BallPosition/out";
    QString RodInputAdress[8];

    for(int i=0; i<8; i++) {
        RodInputAdress[i] = table_robot_adress_prefix + "RodController" + QString("%1").arg(
            i) + "/in";
    }

    Network::init();

    PositionCommunicationPort.open((table_local_adress_prefix + "/in/position/").toAscii().
        data());
    connected = Network::connect(PositionOutputAdress.toAscii().data() ,
        PositionCommunicationPort.getName().c_str());

    BallPositionCommunicationPort.open((ball_local_adress_prefix + "/in/ballposition/").
        toAscii().data());
    connected &= Network::connect(BallPositionOutputAdress.toAscii().data() ,
        BallPositionCommunicationPort.getName().c_str());

    for(int i=0; i<8; i++) {
        RodCommunicationPort[i].open((table_local_adress_prefix + "out/rodcontroller" +
            QString("%1").arg(i) + "/").toAscii().data());
        connected &= Network::connect(RodCommunicationPort[i].getName().c_str(),
            RodInputAdress[i].toAscii().data()); // local, external
    }

    return connected;
}
```

Listing 4: Yarp Server Port initializing

A.8 Sending bottles via Yarp Server

```
// Move rod to a certain position
void MorseClient::MoveRod(int rod_index, float pos, float rotation) {
    Bottle &cmdBottle = RodCommunicationPort[rod_index].prepare();
    cmdBottle.clear();
    cmdBottle.addDouble(pos);
    cmdBottle.addDouble(rotation);
    RodCommunicationPort[rod_index].write();
}
```

Listing 5: Send reference signals via Yarp Server

A.9 Receiving bottles via Yarp Server

```
    // Catch bottle of the Rod Positions
    Bottle *PositionBottle;

    PositionBottle = PositionCommunicationPort.read(false);

    if(PositionBottle != NULL) {
        // Update the positions of the rod
        for(int i=0; i<8; i++) {
            rods[i].position = PositionBottle->get(i).asDouble();
            rods[i].rotation = PositionBottle->get(i+8).asDouble();
        }
    }

    // Catch bottle of the Ball Position
    Bottle *BallPositionBottle;

    BallPositionBottle = BallPositionCommunicationPort.read(false);

    if(BallPositionBottle != NULL) {
        // Update the position of the ball
        ball_real.x = BallPositionBottle->get(0).asDouble();
        ball_real.y = BallPositionBottle->get(1).asDouble();
        ball_real.z = BallPositionBottle->get(2).asDouble();
    }
```

Listing 6: Receive Rod and Ball positions via Yarp Server

A.10 Yarp Communication test

```
    int main()
{
    BufferedPort<Bottle> Port;
    Network::init();
    Port.open("/out");
    Network::connect("/out" , "/in");

    for(int i=0; i < 10000; i++) {
        Bottle &bot = Port.prepare();
        bot.clear();
        bot.addInt(QTime::currentTime().msec());
        Port.write();
        usleep(100);
    }

    return 0;
}
```

Listing 7: Sender

```
int main()
{
    BufferedPort<Bottle> Port;
    Network::init();
    Port.open("/in");
    Network::connect("/out" , "/in");

    ofstream myfile;
    myfile.open ("10000hz.txt");


    int time_rec;
    int time_cur;
    int delay;

    int start_time = QTime::currentTime().second();
    cout << start_time;
    cout << "\n";

    int i = 0;

    for(;;) {
        if(QTime::currentTime().second() - start_time > 15) {
            break;
        }
        Bottle *bot;
        bot = Port.read(false);
        if(bot != NULL) {
            time_rec = bot->get(0).asInt();
            time_cur = QTime::currentTime().msec();
            delay = time_cur - time_rec;
            if(delay < 0) {
                delay = 1000+delay;
            }
            myfile << delay;
            myfile << "\n";
            i++;
        }
    }

    myfile.close();

    return 0;
}
```

Listing 8: Receiver

A.11 Sending bottles for MORSE-Client communication test

```
import GameLogic
import morse.core.sensor
import bpy
import math
from datetime import datetime

class TestClass(morse.core.sensor.MorseSensorClass):
    """ Class definition for the test sensor.
        Sub class of Morse_Object. """

    def __init__(self, obj, parent=None):
        """ Constructor method.
            Receives the reference to the Blender object.
            The second parameter should be the name of the object's parent. """
        print ("######## TEST-SENSOR INITIALIZING ########")
        # Call the constructor of the parent class
```

```
17          super(self.__class__,self).__init__(obj, parent)
18          self.local_data['time'] = 0
19          self.local_data['i'] = 0
20          self.local_data['start'] = 0
21          self.local_data['starttime'] = datetime.now().second
22          print ('######## TEST-SENSOR INITIALIZED ########')
23
24      def default_action(self):
25          """ Main function of this component. """
26          if(self.local_data['start'] > 300):
27                  if(self.local_data['i'] < 10000):
28                          dt = datetime.now()
29                          time = round(dt.microsecond)
30                          self.local_data['time'] = time
31                          self.local_data['i']+=1
32                  else:
33                          print("done")
34          else:
35                  self.local_data['start'] += 1
36                  print("waiting")
```

Listing 9: Morse Test Sensor