Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

An application programming interface for the MORSE simulator

Lukáš Bertl
Cybernetics and Robotics: Systems and Control
bertlluk@fel.cvut.cz

January 2017
Supervisor: RNDr. Miroslav Kulich, Ph.D.

Acknowledgement / Declaration

I would like to express my gratitude to my supervisor RNDr. Miroslav Kulich, Ph.D. for a great mentorship, patience and wise comments that helped me complete this project.

I would like to thank my girlfriend and my parents for their unlimited mental support throughout my whole studies.

Finally, I thank my brother and Kačka Janatková for the proofreading of this thesis.

I hereby declare that I have completed this thesis with the topic "An application programming interface for the MORSE simulator" independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, January ...., 2017

....

Lukáš Bertl

Abstrakt / Abstract

Práce představuje CCMorse, což je knihovna pro komunikaci se simulátorem, kterou jsem vytvořil. Práce dále popisuje proces vývoje simulačního pro simulátor MORSE.

Teze probírá nejprve teorii robotických middleware a robotických simulátorů. Dále obsahuje stručné popisy různých simulátorů, které jsou potom porovnány za účelem simulování mobilních robotů, následované popisem knihovny CCMorse umožňující uživateli psát klientskou applikaci v jazyce C++ za použití rozhraní Player a spustit jí v simulátoru MORSE. Ke konci text popisuje začlenění simulátoru do existujícího systému, ilustruje proces vytvoření simulované reprezentace systému a ukazuje praktické použití knihovny v kurzu zaměřeném na mobilní robotiku.

Knihovna CCMorse může být díky své vícevrstvé struktuře použita i k implementaci jiného rozhraní než Player rozhraní a také může být upravena pro podporu dalšího simulátoru mimo MORSE.

Klíčová slova: mobilní robot, simulace, e-learning, výukové technologie

Překlad titulu: Aplikační rozhraní pro simulátor MORSE

Thesis presents the CCMorse, a simulator communication library, that I have created. The thesis also describes the development process of a MORSE simulation environment.

The thesis discusses the theory of robotic middlewares and robotic simulators first. It contains a brief introduction to various simulators which are compared for the purposes of simulating mobile robots, followed by a description the CCMorse library that enables the user to write a client application in the C++ language using the Player interface and run it in the MORSE simulator. At the end, thesis illustrates incorporation of a simulator into the existing system, it illustrates the creation process of the simulated representation of the system and it shows the practical usage of the library in an introductory course to mobile robotics.

CCMorse can be utilized to implement another API on top of the Player and it can be also altered to support another simulator besides MORSE, thanks to its layered structure.

Keywords: mobile robot, simulation, e-learning, learning technologies

/ Contents

Tables / Figures

Chapter 1
Introduction

Robotics is getting out of the laboratories and heading to become a part of everyone's life. Near future will bring high demand for robotics specialists across industries, so universities are supporting its robotics study programmes by development of various learning systems of robotics for their students. This also provides a platform for research in this specific field. The SyRoTek system is one of them and it is used to teach and study mobile robotics at Czech Technical University in Prague.

Because inexperienced students train their skills on such systems, the simulated copy of the system is needed for safety of the often expensive hardware and to speed up development of a robotic application. Students can evaluate their programs in the simulation and then fine-tune the application in the real system.

Robotic simulators are getting faster and more realistic which corresponds with a rapid increase of computing power of personal computers. SyRoTek has used the Stage simulator that is no longer being developed or supported. Therefore, it has been decided to select a new simulator. The new simulator should be actively developed, it should provide a physically as well as visually realistic simulation in 3D and it should be easily incorporated into the existing system.

The goal of this thesis is to show the process of developing the SyRoTek system simulation environment, and to provide an overview of the CCMorse communication library, that I have created. The CCMorse library has been designed to create an application interface between the existing system and the new simulator, so the already written algorithms will be still usable with no or with only minor changes.

1.1 Thesis outline

Chapter 2 includes multiple topics. The SyRoTek system is introduced in Section 2.1 followed by a theory around robotic *middleware* and two middlewares demonstrated in Section 2.2. An introduction of several robotic simulators, their comparison and the selection of a new simulator for SyRoTek are located in Section 2.3. Features of the selected simulator are outlined in Section 2.4. The philosophy behind the CCMorse library is explained in Chapter 3. The process of developing a simulation environment is shown in Chapter 4, together with highlighted features of developed components, and demonstration of a set of simulation templates created for the Practical Robotics course. An evaluation of the work and the outlook for the future work are contained in the last Chapter 5.

Chapter 2
Current state

This chapter looks into the current state of robotics simulation related systems and software. The SyRoTek system is introduced first followed by an explanation of what robotic middlewares are. Two middlewares are introduced. Several robotic simulators are presented, and their pros and cons are discussed.

2.1 SyRoTek

SyRoTek [1] (”System for Robotic e-learning”) was developed at Czech Technical University in Prague (CTU), Prague, Czech Republic, and it provides the ability to test and learn various mobile robotics algorithms created by students or scientists in a real, but closed and safe environment (for persons as well as for robots). The whole SyRoTek system consists of an arena, a group of mobile robots, web pages [2] with reservation, observational and educational interfaces, cameras for positioning and observation purposes and a main control server. The user can connect to the main server via Internet to control the robot with a program while watching a live video stream from the arena to observe if the program works in real life.

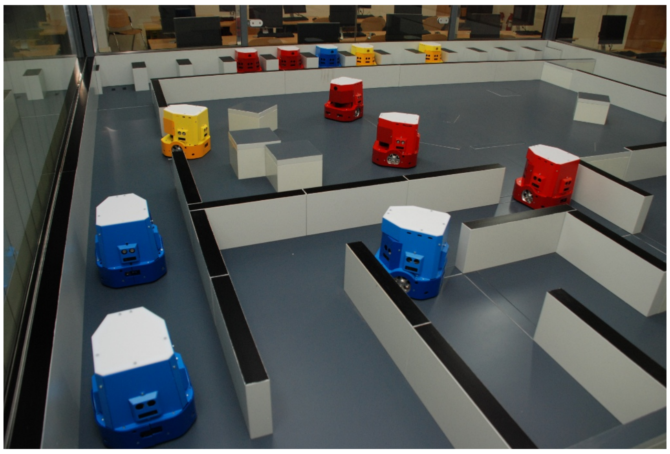

Figure 2.1. A view of SyRoTek arena with robots. Taken from [2].

The SyRoTek arena shown in Figure 2.1, provides charging docks for robots, cameras for observation in a maze-like environment that can be dynamically changed or divided without manual manipulation with the arena. This provides a unique and safe area where up to 13 robots can move.

The SyRoTek robot called ”S1R” (also shown in Figure 2.1) is a two-wheel mobile robot with differential control and can be equipped with a laser rangefinder or an array of ultrasound sensors (sonars). It also carries an array of infra-red range sensors, cameras, and few other sensors. The robot can use odometry or a positioning system built in the SyRoTek arena to determine its position.

2.2 Robotic middlewares

The following paragraphs are focused on robotic middleware, the functionalities they provide and also introduction of the two commonly used middlewares. Next, the focus is turned to robotic simulators. A simulator that has been used with SyRoTek is introduced along with the description of the requirements for a new simulator. The proper simulator is selected by comparing several available simulators and thus brings us to the purpose of the CCMorse library.

The open encyclopedia [3] defines middleware as follows:
"*Middleware is computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue".* "

A more robotics related definition from [4] can be useful where middleware is defined as follows:
"*Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system.* "

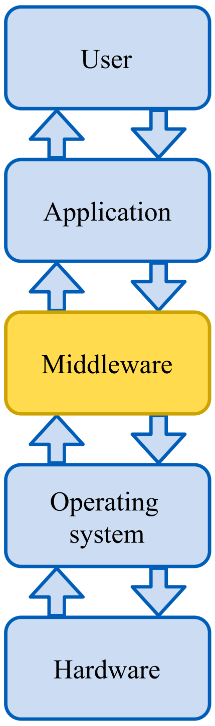

Figure 2.2. Location of middleware in the system. Inspired by [5].

Middleware is used in robotics to hide the low-level implementation of each hardware component of the robot behind it and to create a higher-level abstraction of devices. This leads to better portability of programs, simpler software design, lower development cost, faster testing and easier learning. In Figure 2.2 the position of the middleware in the system can be observed. Middleware takes commands from the application and prepares them for a specific hardware. This means that the user can change the hardware part and their program will still work without a modification. Two different robotics middlewares are presented in following sections.

2.2.1 Player

As can be percieved in [1], SyRoTek implements the Player interface from the open-source project Player/Stage [6–9], which originated at the University of Southern California, California, USA. The Player framework is based on a client-server topology with a queue-based message system. The principal aim of the Player [5] is to provide a development framework supporting different hardware devices and common services needed by various robotic applications and transferring a controller from simulation to real robots with as little effort as possible.

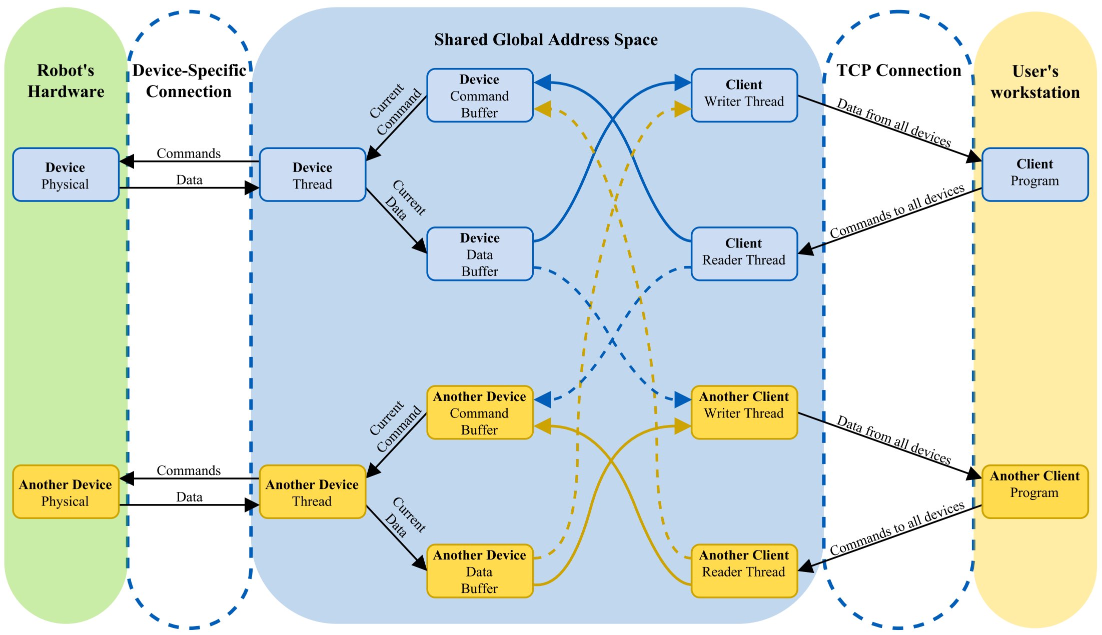

Figure 2.3. Overall system architecture of Player. Inspired by [6].

As can be seen in Figure 2.3, the Player server enables multiple clients to control multiple devices, which can be mounted on multiple robots, via a TCP socket. Player is designed as a heavy multi-threaded server, which reflects the need for simultaneous support for many heterogeneous devices and many heterogeneous clients. This means that data can be obtained from and commands can be sent to each device at the highest possible frequency, which leads to maximisation of the responsiveness of the system.

After the start of a Player server, the server listens to a given port (typically 6665), and awaits connection from a client application. Then, the client subscribes to devices. Subscribed devices will provide data needed by the client, or they will be ready to accept commands from the client. After this initialization, the client application can exchange data with devices. The Player framework introduces a triple of files that provides an abstraction of a device for a client application – a *driver*, an *interface* and a *device* [8]. The driver supports a specific piece of hardware, such as a SICK LMS200 scanning laser rangefinder. The interface provides a generalised set of tools or functions for a piece of hardware, for example "a laser rangefinder" and the device specifies a pair of driver and interface, e.g. a SICK LMS200 driver and a *laser* interface. Because the client application uses Player interfaces to communicate with devices, the application can run in the simulator or on a real hardware with no changes to the client program.

Player was designed as code language independent. It provides several client libraries, such as C++, Java, Python, and Tcl. Application programming interface (API) of these libraries is relatively easy to learn and use. Furthermore, Player makes no assumptions about how the user might want to structure his/her client application. It can be a simple loop or a highly multi-threaded program.

A Player client library implements a set of *proxy* classes. A proxy class offers the functionalities of Player interfaces mentioned above to the user, in a specific programming language. For example, `Postion2dProxy` from the Player's C++ client library PlayerCc can be used to control a position of a robot with a C++ client application.

One of the negatives of Player can be impossibility to determine whether the data received from the sensor are "fresh" or not. The user only knows, that the data are the latest data in the sensor. This can lead to reading some of the data multiple times and treating them as fresh. For example, the user can have the client application reading data from sensors with a frequency of 10Hz, but the user-controlled robot has a laser rangefinder operating on 5Hz. This will result in a situation, where the user's program reads same data from the laser about two times.

2.2.2 ROS

Another middleware, also called the "meta-operating system", is quite commonly used ROS [10–11] ("Robot Operating System") which consists of many small tools designed to work together [5]. It was originally started as a part of a project at Stanford University, California, USA. Now, it is an open-source project with many contributors from all around the world. ROS can be used as a middleware for controlling a real robot or it can be utilised with many simulators, which support ROS interface (will be further discussed in the Section 2.3). ROS is equipped with RViz ("ROS visualisation"), which can be used for data visualisation. ROS "provides operating system's services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management" [10].

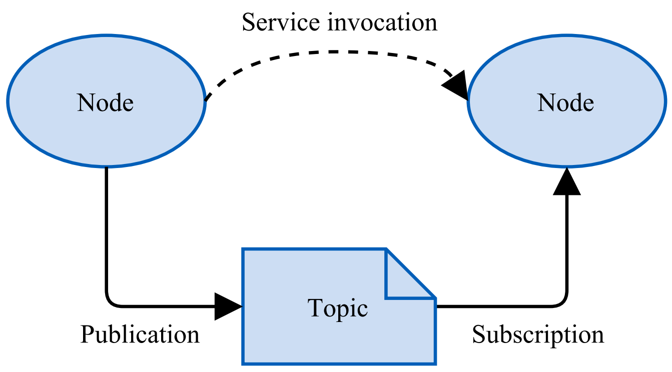

Figure 2.4. Concept of communication between nodes in ROS. Taken from ROS Wiki[1].

ROS uses peer-to-peer communication between "nodes", representing a system as "a computation graph consisting of a set of nodes communicating with one another over edges", where nodes represent individual software packages – devices that can communicate. For example, implementation of a proximity sensor can communicate directly with an actuator of one of the wheels. The ROS interface consists of nodes (software modules), messages (passed peer-to-peer), topics, and services (analogous to web services). Each node can publish data to a specific topic by sending a message.

[1] `http://wiki.ros.org/ROS/Concepts`

Messages are then sent to nodes that are subscribed to that topic. Messages are programming language independent, and a topic is simply a specification of the data. The communication process is described in Figure 2.4.

ROS is one of the most commonly used robotic platforms because of its wide applicability. The main drawback of ROS is that it has a steep learning curve[1] due to its complex and maybe confusing nature.

SyRoTek has the ability to be controlled by both Player and ROS, but using Player interface is recommended over the ROS for beginners because Player is easier to use and learn, which is important for students with different and often limited programming experiences.

2.3 Robotic simulators

As with any other software, it is good to evaluate that the program does what is expected to and does not represent a threat to the system where it is running. Robotics is no exception. Thus, a simulator is needed.

There are several free, open-source or commercial robotics simulators on the Internet from which we can choose from. These software packages differ in various aspects like communication topology, abstraction level, physics engine, graphics engine or other software needed for simulating robots.

Stage [7] is a 2D multi-robot simulator, it was developed within Player/Stage project. It can simulate a big number of simplified robots in flat indoor environments. Stage was used to simulate SyRoTek for several years, but the Player/Stage project is no longer being developed (last released version is from 2010 [9]). therefore, it was decided to find a new simulator that fits specific needs for simulation in SyRoTek.

Criteria for an optimal SyRoTek simulator that will replace Stage are listed here:

- Easy incorporation to existing SyRoTek architecture
- Free of charge for research purposes and students
- Support of existing source codes of the algorithms with no or very small changes
- Simple interface and easy-to-work with functions

Player can be also used with Gazebo [12], it is a 3D open-source multi-robot simulator. Initially, it was developed as a part of the Player/Stage project, but then it became an independent project. Gazebo was developed as an outdoors simulator. Gazebo uses the ODE [13] ("Open Dynamics Engine") as physics engine, but it can be also switched to Bullet [14] (Bullet is state-of-the-art physics engine developed in part of Blender 3D [15] modelling software.), Simbody [16] or to DART [17] ("Dynamic Animation and Robotics Toolkit"). It uses OpenGL as a 3D visualisation engine. Gazebo can accommodate around ten robots, whose bodies are constructed from boxes, spheres, cylinders, planes, and lines, that are joined with joints. While this high-abstract robot model might be sufficient for various robot applications, a creation of a detailed model could be difficult. The user can control joints that can represent actuators. Gazebo also provides a library of pre-made sensors for a robot. In the newer version, Gazebo is operated through graphical user interface (GUI), where the user can see and edit simulation environment and robot model.

Another simulator based on OpenGL and ODE engines is a commercial robot simulation package called Webots [18], developed by Cyberbotics company founded in 1998

[1] Steep learning curve means that it takes a lot of knowledge before someone is able to accomplish something with the tool, contradictory to how the curve actually looks.

as a spin-off from the MicroComputing and Interface Lab (LAMI) of the Swiss Federal Institute of Technology, Lausanne (EPFL), Switzerland and is still actively being developed. Webots can also serve as a middleware, and it is oriented on fast prototyping and simulation of mobile robots. Webots provides a large robot, sensors, actuators and environment libraries with modelling and programming GUI. However it does not allow to create custom robots or devices with a free license. It would not be thus possible to use Webots with the free licence because the simulation of SyRoTek's unique S1R robot is needed.

A next available simulator is V-REP [19] ("Virtual Robot Experimentation Platform"), developed by Coppelia Robotics company, located in Zürich, Switzerland. While V-REP is free to use for students and universities, it is not an open-source project, so it does not grant 100% control over the simulation. V-REP is created to be portable, flexible and versatile. V-REP also uses the ODE physics engine as Gazebo or Webots but can be switched to Bullet [14] or Vortex [20]. V-REP is operated from GUI, which is more user-friendly than Gazebo's control. Because V-REP uses its own graphic engine, it could run smoother on some machines, but it does not provide such photo-realistic visualisation as MORSE for example.

The last considered simulator is MORSE [21–23] ("Modular Open Robots Simulation Engine"), jointly developed at Laboratoire d'Analyse et d'Architecture des Systèmes–Centre national de la recherche scientifique(LAAS–CNRS) and ONERA, both located in Toulouse, France, has now over 20 contributors from the academic environment. MORSE is based on Blender 3D [15] software using its Blender Game Engine (BGE) for photo-realistic 3D visualisation and Bullet [14] for simulating physics. Blender is an open-source multi-platform application focused on creating 3D models, rendering, computer generated animation, post-production activities and last but not least creating interactive applications. Because of that, MORSE does not need GUI and it is only operated from a command line. MORSE supports many middlewares and communication protocols, sadly for us excluding Player. Models of robots or environments are created with Blender, and it is up to the user, how much fidelity they want to bring to the model. Furthermore, Blender supports many formats of 3D models, so it is possible to import almost any model. MORSE provides the ability to select a degree of realism of the simulation, for example, the user can choose what data will be measured by a laser rangefinder and whether data contain a signal noise.

An overview of various aspects of simulators described above is summarised in the Table 2.1.

Simulator	Licence	Graphics engine	Physics engine
Stage	GNU GPL	Internal	Internal
Gazebo	Apache 2.0	OGRE[1] (OpenGL)	ODE/Bullet/ Simbody/DART
Webots	Proprietary	OGRE (OpenGL)	Customized ODE
V-REP	Prop./GNU GPL	Internal	ODE/Bullet/Vortex
MORSE	BSD	Blender Game Engine	Bullet

Table 2.1. Table comparing various attributes of robotics simulators.

2.3.1 Comparison of simulators

[1] Object-Oriented Graphics Rendering Engine: `http://www.ogre3d.org/`

The new simulator for SyRoTek will be chosen in this section. In Table 2.2 virtual ”points” has been assigned for every above mentioned simulator and for the criteria set out on page 6. As it is evident, there is no perfect solution. Yet, there are two simulators that got three out of four points in suitability for simulating SyRoTek: Gazebo and MORSE.

Because Gazebo started as part of Player project, it is the only simulator still supporting Player communication.

When it comes to the availability, Webots received no points because it is not free, compared to the other simulators which can be used for free, at least at the university.

While a GUI can be easy to use by many users, it requires the user to learn its layout and general behaviour of the application. Therefore, a point was given only to MORSE because the whole simulation environment can be prepared in a few easy-to-edit scripts. Then the whole project can be exported with a zip archive to any machine with MORSE. Then, it takes only one command to start the simulation.

Points were given to Gazebo, ROS and MORSE for extensibility because they all are open-source projects, so a creation of a new component with specific needs, can be started from an existing component just by altering its parameters, which is a nice feature that contributes to rapid development.

Simulator	Player support	Free	Easy usage	Extendible
Gazebo	Yes	Yes	No	Yes
Webots	No	No	No	No
V-REP	No	Yes	No	No
MORSE	No	Yes	Yes	Yes

Table 2.2. Table showing suitability of simulators for simulating SyRoTek.

MORSE was found to be the winner after considering all all features discussed above and with the addition of MORSE’s easy of use interface, which is a key feature for students working with SyRoTek. The only feature that MORSE is missing is built-in support for Player interface. That is the essential reason why was CCMorse library developed. CCMorse enables communication between MORSE and programs written for Player with no or only little changes in code.

2.4 MORSE simulator

In this section, the MORSE simulator will be reviewed in more detail. MORSE [21–23] is a generic simulator focusing on academic robotics. It is developed to simulate the robot as whole, in target environment without the intention to replace dedicated simulators for very specific purposes. MORSE can simulate from small to large, indoors or outdoors environments with medium to high level of abstraction. MORSE can simulate from one up to several tenths of robots, depending on the number of sensors and actuators on each robot and their frequencies.

MORSE main design element is enabling the ability to select the level of realism (or abstraction) of simulation to the user. For example, if the user is working on a camera vision, he/she would need a precise model of the cameras, but because he/she does not care about robot’s movement control, hence he/she would be happy with position control of the robot. This also means, that when the user simulates a realistic (non-abstract) scenario, a simulation should produce the same kind and amount of data

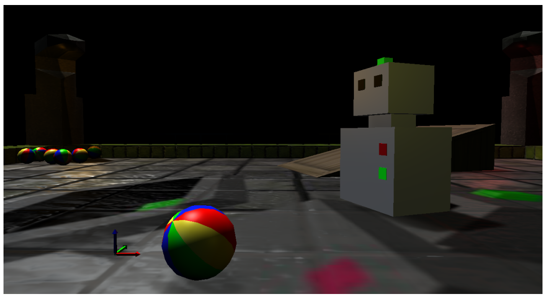
Figure 2.5. Screenshot of indoor simulation in MORSE with Morsy.

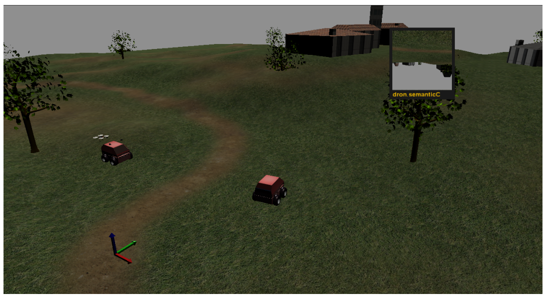

Figure 2.6. Screenshot of outdoor simulation in MORSE with two mobile and one aerial robot.

as would the sensors on the real robot. Similarly, if a more abstraction is used, the simulation accepts higher level commands and can produce more abstract data.

As mentioned in Section 2.3, MORSE extensively uses Blender 3D [15] for simulation. Even though Blender is not designed for simulation, it has multiple properties, that can support a development of a simulator. Thanks to Blender's photo-realistic rendering, MORSE can accurately simulate vision-based sensors, such as stereovision. Meaning that, the user can use the same algorithm on the virtual image as on real images with very similar results. Also, the user can place multiple cameras into the simulated scene and observe multiple aspects or angles of the scene through them.

Every individual module for MORSE such as Pioneer3DX robot, GPS sensor, simple motion actuator, or any custom component, consists of two parts: a Blender file and a Python script. The Blender file holds, besides a 3D model of the component, also information about the physical properties like mass and friction of the object, its bounding box for collision detection, component's role relative to another object in simulation and other analogous variables. In addition to that, a Blender file defines so-called *Logic Bricks*, that allows the Python script to interact with objects in the scene via dedicated API. This mechanism binds methods implemented in the Python script to certain events generated in the simulation.

The Python scripts of all components are based on abstract class `MorseObjectClass` that has information about a 3D position and orientation of the object relative to the origin of the Blender world, `local_data` dictionary with data, which object share with others and must implement the `default_action` method that determines the functionality of the component. As indicated in the beginning of this paragraph, there are three kinds of robotic components available in MORSE.

- Sensors – Gather data from the simulated world, by emulation of the behaviour of a real sensor.
- Actuators – Devices that affect their own state or the state of the whole robot, typically move the robot to a given position or with given linear/angular velocities.
- Robots – Robot component functions as platforms for sensors and actuators. Defines physical properties of the robot and can define robot specific methods.

On top of those three, MORSE features another three additional simulation components.

- Environments – Model world, where the robot exists in a simulation, can be used any 3D model of the environment, both artificially created or captured in the real environment.
- Middleware – Represents communication channels that can be established between simulation and user's program.
- Modifiers – Special modules that alter sensor data to be more realistic, typically by adding noise to data.

MORSE supports multitude of middlewares, namely Socket, ROS [11], YARP [24], Pocolibs [25], MOOS [26], MAVLink [27] and HLA. For SyRoTek simulation, Socket is the most interesting middleware. MORSE shares data through a socket as a straight serialisation of the JSON [28] ("JavaScript Object Notation") representation of the component data.

Chapter 3
CCMorse library

In the following chapter, the CCMorse library will be discussed.

As already mentioned, SyRoTek uses Player API, a feature we want to preserve. MORSE was chosen as the new SyRoTek simulator, but because MORSE does not support the Player API, the creation of an interface that would allow us to try out programs written for SyRoTek with Player API on MORSE was needed. For this purpose, the CCMorse library was created. CCMorse library allows control of the MORSE simulation with algorithms written as Player applications. As can be seen in Figure 3.1, the user decides if he/she wants to compile his/her program for simulating on MORSE or to test it in the real SyRoTek arena. CCMorse imitates the same interface as a part of Player proxies which are needed for SyRoTek simulation. So user's code can be compiled for SyRoTek as well as for MORSE without the need to change the code.

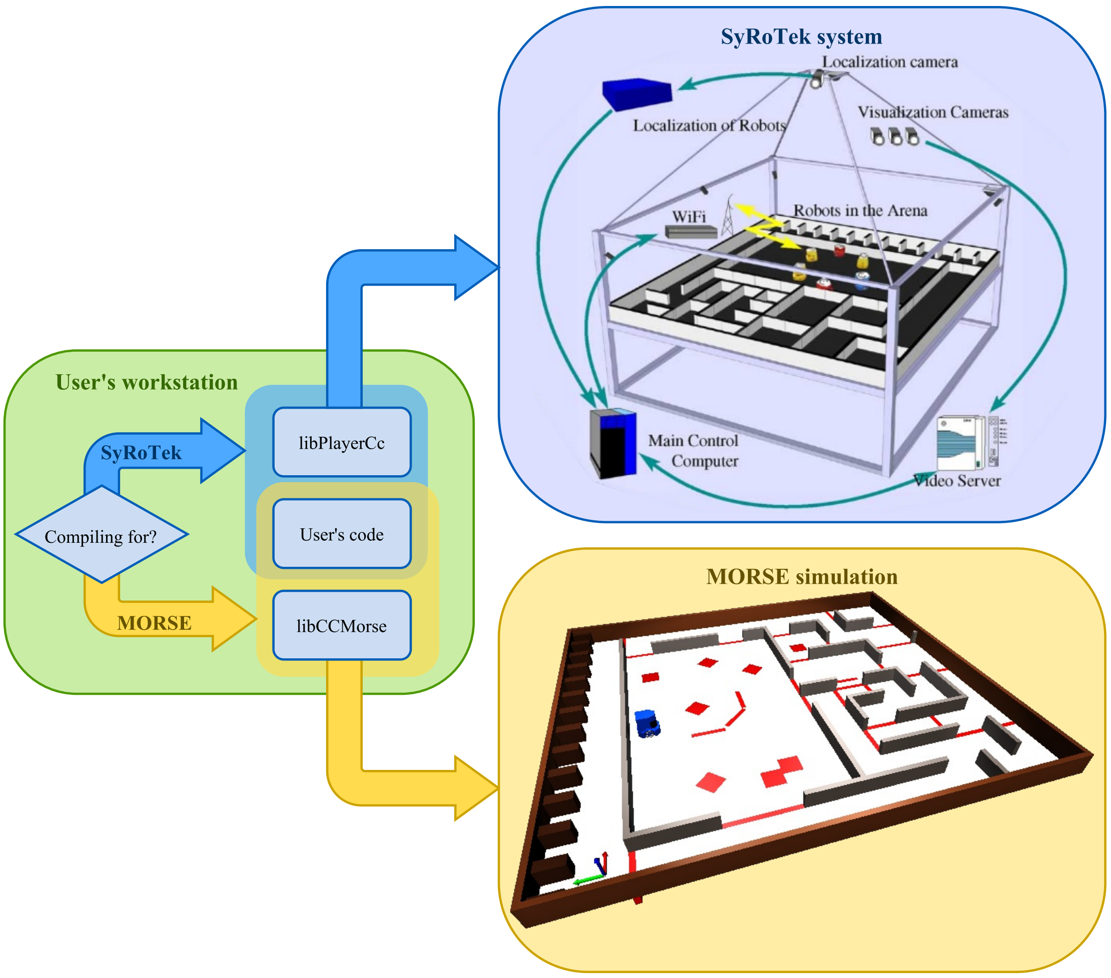

Figure 3.1. Diagram showing of CCMorse library usage. SyRoTek part taken from [1].

The CCMorse library was created mainly for use with simulated SyRoTek, but it can serve for other simulated worlds too. Due to its purpose, CCMorse library implements only a handful of devices, and those are devices mounted on the S1R robot: laser rangefinder, position sensor, odometric sensor, and motion controller. Possible extension of compatible devices should be relatively easy, and this topic will be discussed in the next Section 3.1.

3.1 Philosophy

CCMorse is written in C++ and it incorporates all benefits of object-oriented programming. Library is only dependent on Boost C++ library [29] which is used only for managing multi-threading in client applications. The remaining of CCMorse uses only the C++ standard library.

CCMorse is composed from three main layers: *Core*, *MORSE-like API* and *Player API*. These layers are stacked, and each layer utilises methods of those layers beneath it as Figure 3.2 shows.

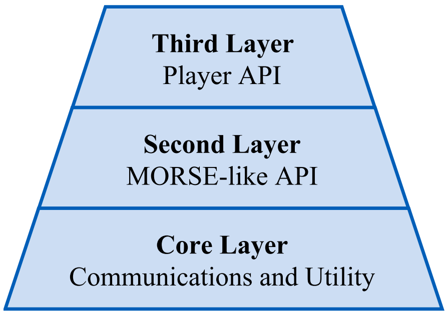

Figure 3.2. Diagram showing of CCMorse's program layers.

3.1.1 The Core layer

The core layer contains a basic set of methods for socket communication, JSON data deserialization, exception handling, basic data manipulation and it also defines useful variable types. The Core layer further comprises of main `MorseClient` class, `MorseRobot` class and `MorseDevice` base class representing a component or other entity in MORSE simulation, that stores data and can share them.

The `MorseClient` class is implemented as a *singleton* and it stores essentially all simulation information and data. Upon creation, the `MorseClient` connects to a simulation and obtains a list of all devices and all robots in the simulation. Then the `MorseRobot` objects are created with their devices accordingly. To ensure that this mechanics works properly, every device class has its set of identifiers or names, that the client will look for in the device list. Therefore, it is convenient to include one of the identifiers into the name of a device in the simulation.

CCMorse supports multi-robot simulation, hence the `MorseClient` class contains a list of all robots in the simulation. The `MorseRobot` class serves as representation of robot in CCMorse. Each instance of the `MorseRobot` class keeps its list of devices that can operate and carries a list of Player's proxies. `MorseRobot` provides quite a few useful robot-oriented methods, from setting the position goal to getting robot's weight.

3.1.2 The Second layer

The second layer is MORSE-like API, which uses the Core layer to implement device-like components in a structure based on the MORSE component library. As Figure 3.3 demonstrates, there are `MorseSensor` and `MorseActuator` abstract classes that act as base classes for sensors and actuators respectively. All devices are implemented on these abstract classes with a similar data structure as components in MORSE. Furthermore, each implemented device adds useful methods that are specific for a given device, for example ”getter” or/and ”setter” functions for individual data fields or function that looks for an angle with minimal measured range with a laser rangefinder.

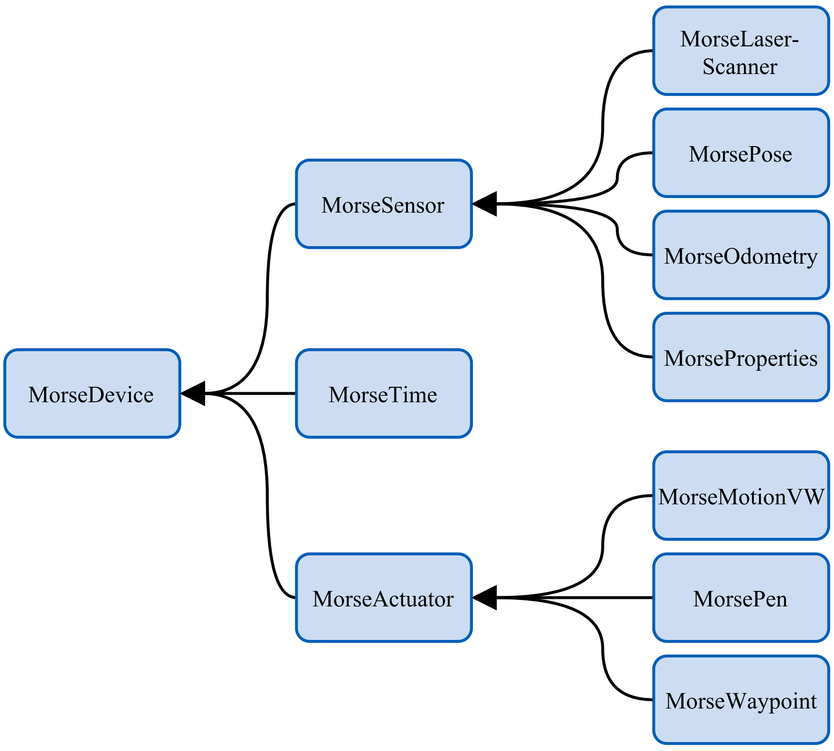

Figure 3.3. Class inheritance diagram of the second program layer of CCMorse. On the right side, there are all implementations of individual components in CCMorse, that are inherited from abstract classes located left of them.

There is also implemented the `MorseTime` class that can receive simulation time and time statistics.

The second layer can be used to control the robot in a MORSE simulation, but because it has its unique interface, i.e. it does not use Player interface, which means that it cannot be used simply with SyRoTek as suggesting in Figure 3.1. This functionality is provided by the third layer. Besides this, the second layer can be used to implement a new API if a replacement of the Player interface (the third layer) is wanted or the CCMorse is used with another project than SyRoTek.

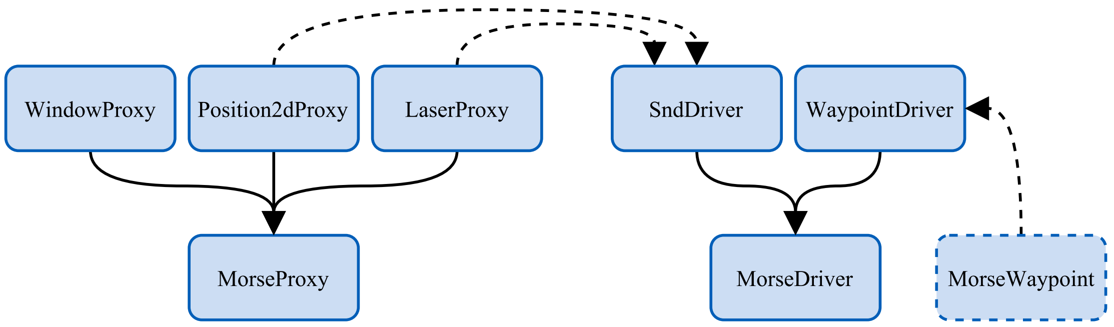

Figure 3.4. Class inheritance diagram of the third program layer of CCMorse. The dashed line indicating the collaboration of classes in the orientation of arrow, i.e. SndDriver uses instances of Position2dProxy and LaserProxy.

3.1.3 The Third layer

The third layer implements Player application interface from Player's C++ client library. This is accomplished with *proxy* classes inherited from `MorseProxy` abstract class, as can be observed in Figure 3.4. Each class in this set provides the same methods and functionality as proxy classes in Player's C++ client library, but the methods are implemented with the second CCMorse layer.

Proxy classes use typically one or more devices defined in the second layer. For example, the `Position2dProxy` class incorporates the functionality of the `MorsePose` position sensor, `MorseOdometry` position sensor and `MorseMotionVW` actuator into one object.

Next Player's feature is a *driver* class, that implements a specific piece of hardware or an algorithm that can take control over the robot. Driver classes in the CCMorse library are based on `MorseDriver` class. When the user starts the main method of a driver class CCMorse creates a new thread where the task is executed. Driver classes borrow functionality from other modules and use them to fulfil more complex algorithms, for example, as Figure 3.4 suggests the `SndDriver` utilises `Position2dProxy` and `LaserProxy` classes to safely navigate robot to goal position through the environment with obstacles.

The last part of Player interface is `PlayerClient` that manages communication with a server in the Player. However, all its functionality is implemented in CCMorse's `MorseClient` class. Thus the `PlayerClient` class just redirects commands to `MorseClient` in CCMorse.

3.2 Configuration

In order to configure proxies and drivers in Player-like manner, a configurable *builder-script* of SyRoTek's MORSE simulation scene has been developed. The simulation builder script reads a configuration file and adjusts the simulation accordingly. The configuration file utilises the JSON notation, for which Python provides a set of powerful tools for simple processing within the script. This script also sets default values to missing items, which means that users can specify only those configurations they need and the rest will be set up by the script. In this way, multiple initial scenarios of a simulation can be arranged by setting these properties. For example, initial position, colour, indexes of proxies available, and configuration of drivers can be specified for each robot. Also, the behaviour of the simulation environment can be set such as placement of the camera in the scene or simulation-time strategy.

These configurations with additional data obtained directly from Blender are then stored in the `RobotProperties`, which is custom MORSE sensor component. This component just provides a platform that enables CCMorse to read Player's specific configurations from the simulation and use them during initialization and runtime of an application. `MorseProperties` class is implemented in the second program layer of CCMorse (Figure 3.3). `MorseProperties` stores settings from `RobotProperties` and provides functions for getting individual entries. With the help of this class, every instance of `MorseRobot` class carries its configuration, accessible at any time during the simulation. Thanks to this mechanism, a new property or configuration whose value does not change during the simulation can be quickly introduced to CCMorse's `MorseProperties` class if needed.

3.3 Communication

As was mentioned above, CCMorse uses socket messages in JSON format to communicate with MORSE simulation. This communication always starts from CCMorse, usually to get new data from sensors and to send new commands to actuators. Sending and reading fresh data to and from all devices is implemented in the `Read()` function within `MorseClient` class, while execution of this function is left to the client application. It is up to the user to decide how he/she wants to acquire the data from the robot, for example, the user can use a simple loop with reading data in the beginning, or he/she can create a separate thread for data reading.

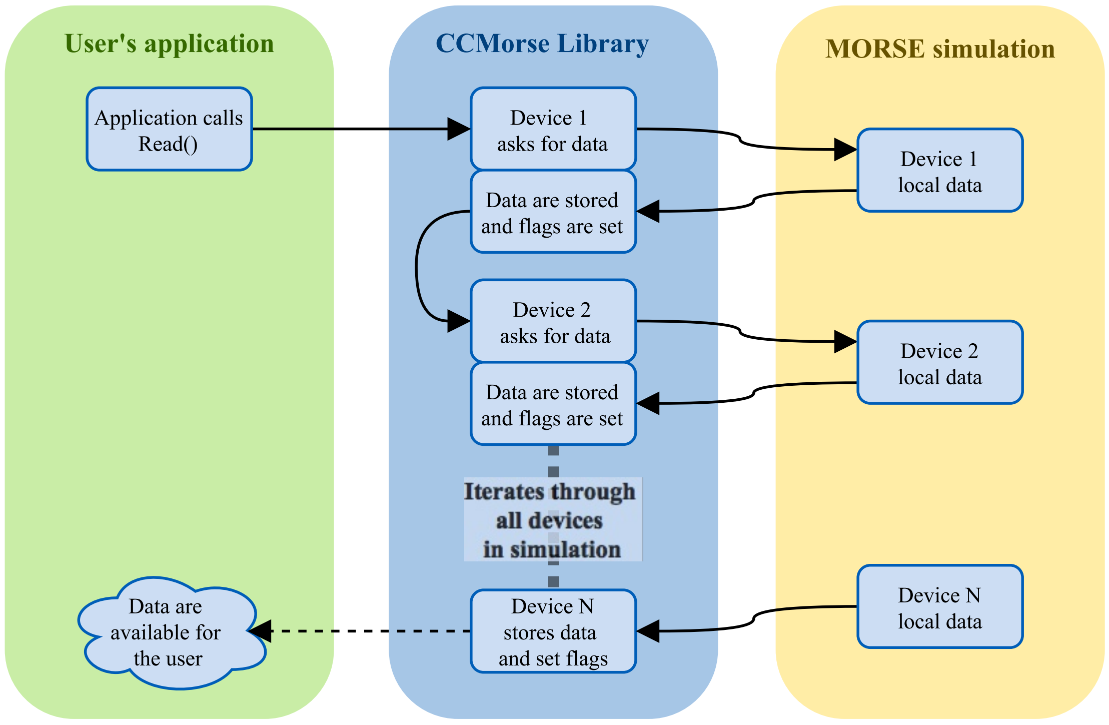

Figure 3.5. Diagram shows how are fresh data obtained through CCMorse from a simulation.

As can be seen in Figure 3.5, when the user calls the read function, CCMorse iterates through each `MorseDevice` and sends a query to get data stored in appropriate MORSE component in the simulation. Then it checks validity of the new data for every proxy.

Valid flag and *fresh flag* are then fittingly raised in proxies. The valid flag is controlled by CCMorse, and the user cannot change its state. The fresh flag, on the other hand, is available for the user and it is set by CCMorse only after acquisition of new data. The user can mark the data as used with this mechanism and avoid using them twice.

3.4 Application example

In this section, an example application will be shown. The application contains robot-control loop making use the CCMorse library. Principles for using the same program in the simulator and in the real system will be explained using the example application.

Although CCMorse has been designed for seamless integration into Player using scripts, some of the components provided by CCMorse need a special treatment in case of using them in the code that will run on the real system. These components are the `WindowProxy` class and the `MorsePen` class, because their real representation does not exist.

Because the target platform of the program is determined during its compilation, it can be specified which parts of the code will be compiled for which platform. Therefore, a Makefile where the user can choose the target platform was developed. The most interesting part of the Makefile is in the following code transcript.

```
 1  # PLATFORM DEFINITION
 2  # - SET THE FOLLOWING TO "YES" TO RUN CODE WITH MORSE,
 3  # - ANYTHING ELSE TO RUN IT ON SYROTEK
 4  MORSE = YES
 5
 6  # Building for MORSE or for SyRoTek?
 7  ifeq ($(MORSE), YES)
 8      CPPFLAGS += -D _MORSE
 9      CPPFLAGS += $(CCMORSE_CPPFLAGS) $(BOOST_CPPFLAGS) $(IMRH_CPPFLAGS)
10      LDFLAGS += $(CCMORSE_LDFLAGS) $(BOOST_LDFLAGS) $(IMRH_LDFLAGS)
11  else
12      CPPFLAGS += $(PLAH_CPPFLAGS) $(BOOST_CPPFLAGS) $(IMRH_CPPFLAGS)
13      LDFLAGS += $(PLAH_LDFLAGS) $(BOOST_LDFLAGS) $(IMRH_LDFLAGS)
14      OBJS += $(OBJSPLAH)
15  endif
```

The user can add this code into his/her Makefile, and they can simply switch between compiling for MORSE or for SyRoTek by alternating the `MORSE` variable (on line 4). When the variable is set to `YES`, the application will be compiled for the use with MORSE, otherwise the program will control the real robot inside SyRoTek. Both platforms differ in libraries which are linked with the code. CCMorse has defined the include path and path to the library itself in variable `CCMORSE_CPPFLAGS` and `CCMORSE_LDFLAGS` respectively (lines 9–10). Similarly the Player paths are included in `PLAH_CPPFLAGS` and `PLAH_LDFLAGS` varibles (lines 12–13).

Please note that the `_MORSE` variable is defined (line 8). With the help of this definition, some parts of the program code can be hidden. The user can enclose parts of code with `#ifdef _MORSE` and `#endif` directives, so these parts will be compiled only for MORSE platform. Alternatively, the user can use `#ifndef _MORSE` directive to compile a selected code for SyRoTek. Furthermore, the `#else` directive can be added for the switching behaviour, as can be seen in next code sample.

The following code is copied from the header file `robot.h` (code listed in Appendix C.1) which is a part of the SyRoTek template project. The template project includes several useful modules which are preventing the reinvention of a wheel. The `robot.h` and `robot.cc` file pair defines the `CRobot` class where the robot-control loop is located.

```
#ifdef _MORSE
    #include "libCCMorse/CCMorse.h"
#else
    #include "libPlayer/Position2dProxy.h"
    #include "libPlayer/LaserProxy.h"
    #include "libPlayer/PlayerClient.h"
    #define PI 3.14159265359
#endif
```

Example of an implementation of a robot-control loop can be seen in Appendix C.2. This code implements a simple obstacle avoiding behaviour of the robot that can be compared to the behaviour of the Braitenberg vehicle. This code can be also used to solve the Introduction task in the Practical Robotics course, more about this experiment will be discussed in Section 4.3.

The short commentary of the `robot.cc` file follows in this paragraph. All code line numbers refer to code listing in Appendix C.2. The method `getConfig` stores the configuration of the program. As can be observed, the right configuration of connection is set by using the `#ifdef _MORSE` directive (lines 16–40). A `PlayerClient` is created using the configured server information, followed by creation of proxies in the `CRobot` constructor (lines 42–71).

The control loop of the robot is implemented in the `navigation` procedure (line 111). Parts of the code where the `WindowProxy` is used are enclosed between directives (lines 114–118 and 161–178). The rest of the code can be used with MORSE as well as with SyRoTek. Lines 146–181 handle the position data. The position is saved, logged and also drawn if the application is compiled for MORSE. Lines 183–268 handle the data from laser.

Laser data are used to compute the straight and angular velocities of the robot. The data are split to left and right halves. It is searched through the current laser scan for a minimal measured range. If the minimal range lies in the left half of the scan, the opposite (right) wheel is slowed down, and vice versa. This is sufficient for a simple collision avoidance. The computed speeds are then set to robot (line 272) and the loop repeats.

Chapter 4
Practical usage of CCMorse

Now, when all the theory about robotics middlewares (Section 2.2), robotic simulators (Section 2.3) and the simulator communication library CCMorse (Section 3) is laid down, the practical usage of MORSE simulator (Section 2.4) in conjunction with CC-Morse to simulate the SyRoTek system (Section 2.1) will be discussed. Additionally, the process of a development of a new component for MORSE simulator and incorporation of such component into the CCMorse library will be demonstrated. Finally, a set of SyRoTek simulation tasks for students attending the *Practical Robotics* course taught at the Czech Technical University in Prague (CTU), Prague, Czech Republic will be presented.

4.1 Creating SyRoTek simulation world

First, creation of models of the S1R robot and the SyRoTek Arena is described in the following paragraphs.

When the user has the MORSE simulator successfully installed on his/her workstation[1] a new simulation (in MORSE called *simulation environment*) can be created. In the terminal, the user navigates to the directory, where he/she wants the simulation project to be located. From this location, the user instructs MORSE to create a new simulation with the `morse create <env>` command, where `<env>` is the name of the simulation environment. Transcript of this procedure in the terminal can be seen in the next block.

```
$ mkdir Simulations && cd Simulations/

$ morse create MorseSyrotek

* A new simulation environment has been successfully created in
</home/user/Simulations/MorseSyrotek>.

* You can run it directly with "morse run MorseSyrotek" or you can start
editing it.
```

If the user runs a new simulation with the `morse run <env>` command, the MORSE default simulation scene will appear. A Morsy robot, the mascot of MORSE, will appear in an environment resembling a playground with several balls and chairs that can be interacted with. This default scene can be seen in Figure 2.5. Also, if the directory, where the simulation environment was created, is inspected, it can be found that MORSE made a new folder with the name of the project. The simulation project consists of a simulation builder-script called `default.py` in which the simulation world is defined and three additional folders *data*, *scripts* and *src* are created. All Blender

[1] The installation of MORSE is described on `https://www.openrobots.org/morse/doc/stable/user/installation.html`.

3D models specific to this project are stored in the *data* directory. The *scripts* folder can contain MORSE's native client applications written in Python, which are not used in this text. Finally, the *src* folder holds builder-scripts and implementation files for components not included in the MORSE component library.

When the MORSE simulation environment is prepared, it is possible to add several SyRoTek specific components needed in simulations. The S1R robot can be added into the simulation environment with `morse add robot <name> <env>` command, where `<name>` is the name of the robot and `<env>` is the name of a simulation environment where the robot will be created. The process of making a new robot called "syrotek" is shown in the following box.

```
$ morse add robot syrotek MorseSyrotek
```

MORSE then creates corresponding template files in the *data* and *src* directories of the project. The further development of S1R robot model for MORSE simulation is commented in Section 4.1.2

In the next step, the *robot properties* sensor, first mentioned in Section 3.2, is added to the simulation environment. For this, the `morse add sensor <name> <env>` command is used, where `<name>` represents the name of the newly created component and `<env>` is the name of a simulation environment where MORSE will create the sensor component. The command presented in the following code block is used.

```
$ morse add sensor robotproperties MorseSyrotek
```

Finally, the last unique component is the Pen actuator which allows us to draw in the Canvas component, both will be further examined in Section 4.2. A new actuator is created with a familiar command `morse add actuator <name> <env>` as can be seen in subsequent instruction, where `<name>` stands for the name of the component and `<env>` is the name of the simulation environment.

```
$ morse add actuator pen MorseSyrotek
```

While the creation of all possible MORSE components was illustrated above, the content of the builder-script is inspected in the following text[1].

```
#! /usr/bin/env morseexec

from morse.builder import *

robot = Morsy()
robot.translate(1.0, 0.0, 0.0)
robot.rotate(0.0, 0.0, 3.5)

motion = MotionVW()
robot.append(motion)

keyboard = Keyboard()
robot.append(keyboard)
keyboard.properties(ControlType = 'Position')

pose = Pose()
robot.append(pose)

```

[1] The comments were erased from the code, to save space.

```
19 robot.add_default_interface('socket')
20
21 env = Environment('sandbox', fastmode = False)
22 env.set_camera_location([-18.0, -6.7, 10.8])
23 env.set_camera_rotation([1.09, 0, -1.14])
```

According to the transcript of the builder file above, the file type is defined (line 1), the `morse.builder` Python module is imported into the script (line 3) followed by the creation of a robot object called `robot` (line 5). It is evident that the `robot` object is an instance of a `Morsy`, which is a default robot in MORSE. The initial position and rotation of the robot in 3D-space are set (line 6–7).

The `motion` object is created as an instance of the `MotionVW` (line 9). The `MotionVW` component enables to control robot's movement with linear and angular velocities. As might be seen, the next command appends the `motion` to the `robot` (line 10), which means that the `motion` belongs to the `robot`, thus `robot` can be affected by the component. Similarly, if a sensor component is attached to a robot, the robot can read data from the sensor. The rest of the script is much the same till the end – a component is created and it is appended to the robot then. It can be seen that components can be configured by changing their properties.

The last thing left is to set up the simulation environment. To do that, the `env` object is created (line 21), the first argument of an environment constructor determines the model of the simulation world. At the very end of the script, the initial location and rotation of the camera are set (line 22–23).

4.1.1 Modelling of the Arena

In the following text, the development of the SyRoTek Arena model will be described in more detail.

The modelling of the SyRoTek's Arena is quite straightforward with a physical description. The length of the Arena is 3.72m, the width is 3.42m, and the height of obstacles is about 16cm.

As it is evident from Figure 4.1, the Arena consists of a white plane serving as the Arena floor, brown borders and dock separators, light-grey blocks representing static obstacles, and red colour of blocks expressing that the blocks are dynamic, and they can be thus extracted or retracted during simulation in the real Arena. In the shown model, all dynamic obstacles are retracted.

In addition to that, the Arena model defines also the position and type of lighting. The Arena is illuminated with a set of four spotlights, that can be seen as black dots in Figure 4.1, and global Sunlight, which is used for casting shadows of models in a particular direction in a simulation.

The Arena can be modelled by utilising only Blender primitive shapes. A plane is used for the Arena ground, and it is stretched up to required dimensions. A material is created and it is then set to the plane object. The bottom-left corner of the plane is positioned at the beginning of the coordinate system. As mentioned above, multiple types of obstacles are used in the Arena. All obstacles are modelled as boxes. The procedure of building walls in the Arena model is similar to modelling of the Arena ground – a box is created, its dimensions are set, and its material is assigned. It is then possible to copy the object of the given type of obstacle and place it appropriately in the model until the Arena is complete. Tutorials and more information about modelling in Blender are available on [30–31].

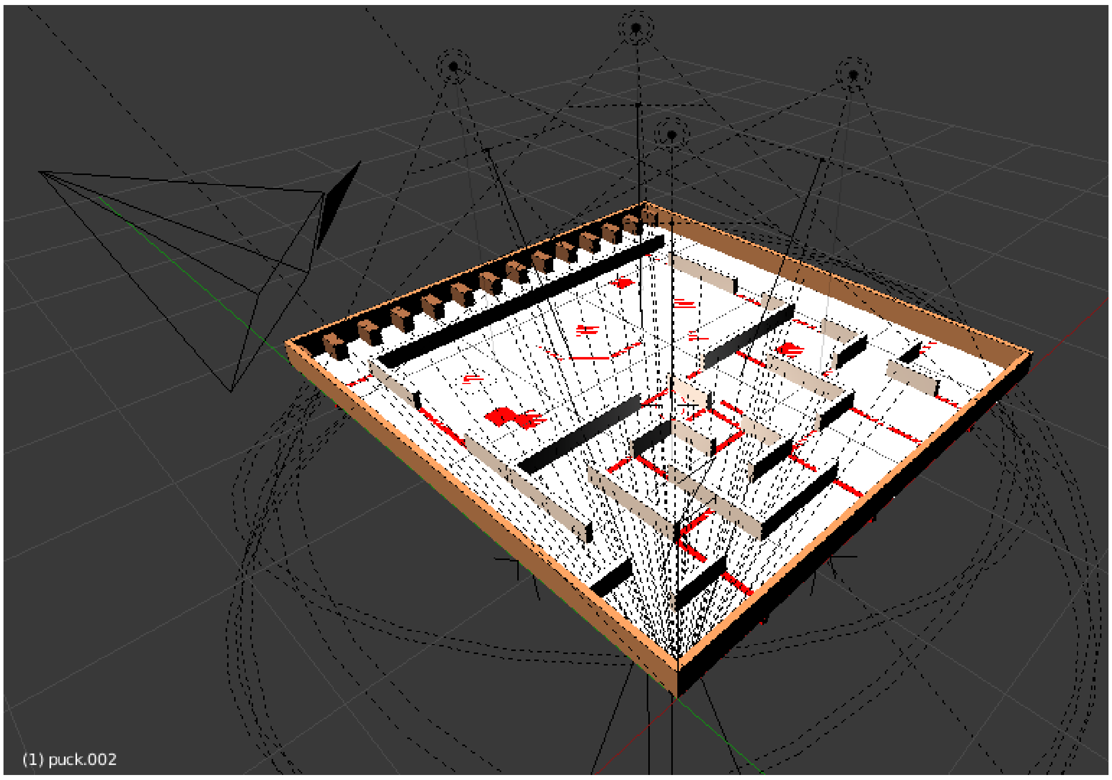

Figure 4.1. Model of SyRoTek Arena in Blender 3D.

When the Arena model is finished, it can be set as a simulation environment in the SyRoTek simulation project. The 3D model file is moved to project data folder. For example, the path relative to the simulation directory is "`./data/MorseSyrotek/environments/arena.blend`". The path to the Arena model needs to be defined in the builder-script. Because Python uses only absolute paths, it is needed to get the path to the project folder. The path can be obtained with a code demonstrated in the following box:

```
1   import os, sys, inspect
2
3   # Getting path to this folder
4   currentFolder = os.path.realpath(
5       os.path.abspath( os.path.split(
6           inspect.getfile( inspect.currentframe() )
7       )[0] )
8   )
9   if currentFolder not in sys.path:
10      sys.path.insert(0, currentFolder)
11
12  # Setting path to environments blend file
13  environmentPath = os.path.join(
14      currentFolder, 'data', 'MorseSyrotek', 'environments', 'arena.blend'
15  )
16  if environmentPath not in sys.path:
17      sys.path.insert(0, environmentPath)
```

This code is added to the top of the builder-script. Modules that are helpful for working with paths are imported at first (line 1). In the next segment (lines 3–10), the absolute path to the root directory of the simulation project is obtained. As can be seen in the lower part of the transcript (lines 12–17), a new variable `environmentPath` is introduced. It stores the absolute path to the Arena model. The constructor of the simulation environment needs to be provided with the path to the world model. The result can look like the command in the following code sample.

```
env = Environment(filename = environmentPath, fastmode = False)
```

4.1.2 Modelling of the S1R

The process of creating a model of the S1R robot will be described in the following text.

When the MORSE component of S1R robot was set up in Section 4.1, a 3D model file was created in the simulation project. The file is located on path "`./data/MorseSyrotek/robots/syrotek.blend`" relative to the root of the project folder. The file contains a model of Morsy. The Morsy model can be deleted in the Blender editor and thus, the modelling of the S1R robot can start. A robot model for MORSE simulation should consist of at least two parts – the collision bounds and the body of the robot. The collision bounds wire-frame of S1R is demonstrated in Figure 4.2. A simulation uses a collision bounds model to determine collisions with other objects in the simulation. Therefore, the collision wire-frame should be as simple as it gets (not including curves), but also it should match the physical dimensions of the robot, in order to run the simulation smoothly and to correspond to the reality. The collision wire-frame does not have to look fancy because it is not visible during the simulation.

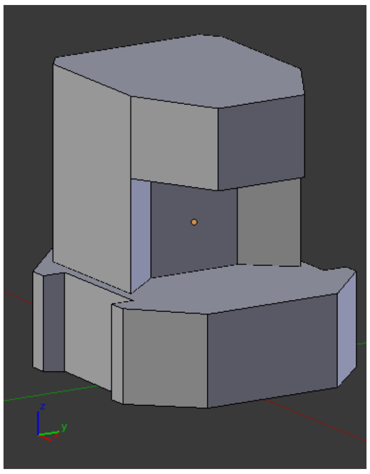
Figure 4.2. Visualized collision bounds of the S1R model in Blender.

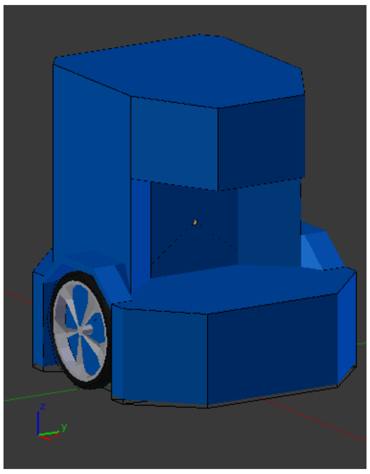
Figure 4.3. Model of the S1R robot with wheels in Blender.

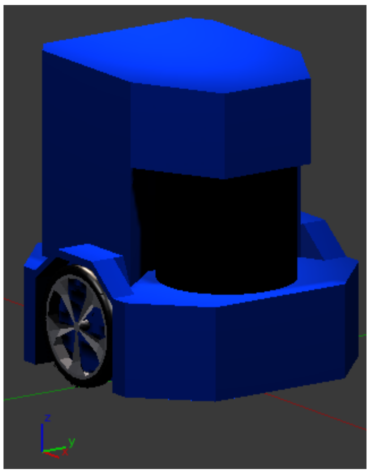
Figure 4.4. Rendered model of the S1R robot with all devices attached in MORSE simulation.

The next part of the robot model is the body. Opposed to the collision bounds, the body represents a visible part of the model, but it has no physical properties. The shaping of the robot body can be started from the collision wire-frame. As can be seen in Figure 4.3, a pair of wheels and wheel fenders are added. Also, the model of the body is lifted about a half of a centimetre above the floor.

The modelling of the robot in Blender can be started with three boxes – a base, a middle part and a top part. The dimensions of all parts are adjusted, and the centre of gravity of the base portion is located at the beginning of the coordinate system. Faces and edges of boxes are deleted, so only vertices are left. More vertices are added accordingly to the model of the S1R robot. The new set of vertices is linked to create edges and edges are then joined to create faces. More vertices can be added to create a more realistic model of the robot. More information about modelling in Blender is available on [30–31].

The model of the S1R robot is ready now and it can be included into the simulation project. The inclusion is accomplished by adding the following code into the builder-script.

```
1  # Setting path to S1R robot folder
2  robotFolder = os.path.join(
3      currentFolder, 'src', 'MorseSyrotek', 'builder', 'robots'
4  )
5  if robotFolder not in sys.path:
6      sys.path.insert(0, robotFolder)
7
8  from morse.builder import *
9  from MorseSyrotek.builder.robots import Syrotek
10
11 robot = Syrotek()
```

The code adds the folder with builder-scripts of robots in system paths (lines 2–6), so the possibility of inclusion (line 9) of a `Syrotek` class into the simulation builder-script is assured. It is now possible to create an S1R robot object in the script by calling a constructor of the `Syrotek` class (line 11).

Because all S1R robots will have the same instrumentation in the simulations, all components can be appended directly in the robot builder-script, it is no longer needed to attach every single device to each robot as was indicated in Section 4.1. When the S1R component builder-script is examined the following code can be seen[1]:

```
from morse.builder import *

class Syrotek(GroundRobot):
    def __init__(self, name = None, debug = True):

        GroundRobot.__init__(
            self, 'MorseSyrotek/robots/syrotek.blend', name
        )
        self.properties(
            classpath = "MorseSyrotek.robots.Syrotek.Syrotek"
        )

        self.motion = MotionVW()
        self.append(self.motion)

        if debug:
            keyboard = Keyboard()
            keyboard.properties(ControlType = 'Position')
            self.append(keyboard)

        self.pose = Pose()
```

[1] The comments were erased from the code, to save space.

```
22          self.append(self.pose)
```

It is evident from the line 5 is evident that the robot constructor takes two arguments – a name of the robot and information whether a debug mode is enabled. The name argument is relevant when multiple robots are simulated because two robots cannot have the same name. If the debug argument is `True`, the user will be allowed to move the robot with the keyboard during a simulation. Robot components are then set (lines 14–23).

The S1R robot has a Hokuyo laser rangefinder that can be added into the robot builder with the following code:

```
# Hokuyo laser scanner (freq: 10Hz, range: 5m, fov: 270deg, samples: 682)
hokuyo = Hokuyo()
hokuyo.translate(0.035, 0.0, 0.0)
hokuyo.rotate(0.0, 0.0, 0.0)
hokuyo.frequency(10)
hokuyo.properties(
    Visible_arc = False,
    resolution = 0.39589,
    scan_window = 270,
    laser_range = 5.0,
    layer_offset = 0.125
)
self.append(hokuyo)
```

The Hokuyo laser rangefinder is available in the MORSE component library, so just properties are set to meet the properties of the laser scanner on the real S1R robot.

An odometric sensor, the `RobotProperties` sensor, a waypoint actuator, and the `Pen` actuator are added in a similar manner as the laser rangefinder. The model of the S1R robot with all components attached looks in the simulation as shown in Figure 4.4. The development of the `Pen` and the `Canvas` component will be described in the next section.

4.2 Developing of a grid-map component

Grid-maps or *occupancy grids* are often used for environment mapping and path planning in the robotics. Visualisation of a grid-map in the simulation is considered to be a useful feature for users. For example, the user can observe if his/hers mapping algorithm works properly or the user can utilise such component for visualising information such as a record of robot's position. The development of the grid-map visualisation component is described in the following passage. Also, incorporation of the component into the CCMorse library is shown.

4.2.1 Component at the MORSE side

The grid-map visualisation component is made of two parts – the `Canvas` component and the `Pen` actuator. `Canvas` is a custom component that can be created in a MORSE simulation, and it is used as visualisation of grid-maps. The `Pen` actuator component is used to draw on `Canvas`, and for that purpose, it implements a number of functions that manage properties of the `Canvas`. For example, the user can draw a point or a line on canvas with a given amount of transparency, the user can alter the resolution of the canvas, or the user can switch the canvas between three view modes.

The Canvas component

The idea behind this two-part mechanism is that there is always only one `Canvas` in the simulation and any robot which owns the `Pen` can draw on the canvas. Therefore the canvas is a passive component used only to display contents of the grid-map. The transcript of the `Canvas` class is shown in the following code block:

```
1  from morse.builder import *
2
3  class Canvas(PassiveObject):
4
5      def __init__(
6          self, filename, alpha = 1, prefix = None, keep_pose = False
7      ):
8          PassiveObject.__init__(self, filename, prefix, keep_pose)
9
10         mat = bpymorse.get_material("CanvasMat")
11         if alpha >= 1.0:
12             mat.use_transparency = False
13         else:
14             mat.use_transparency = True
15             mat.alpha = alpha
```

It can be seen that all builder modules are imported (line `1`) and the `Canvas` class is based on the `PassiveObject` class provided by MORSE (line `3`). The constructor of the class takes four arguments (line `6`). The `filename` parameter specifies the Blender file to load. The `alpha` sets the transparency of the canvas. The `prefix` is searched in the Blender file and only objects whose names are ”prefixed” are loaded in the scene. Last, the `keep_pose` parameter defines whether the object is movable. `PasiveObject` is then created with given parameters (line `8`). In the end, transparency of the canvas is set (lines `10`–`15`). Due to features of the Blender Game Engine, transparency of the canvas cannot be changed after scene initialization.

The model of the canvas is represented in Blender as a square plane with a texture mapped on its surface using a flat projection. So the canvas object acts as a display for the texture content. More details about textures in Blender are available in [31].

The `Canvas` component can be added into the simulation builder-script similarly to obtaining the environment model path in Section 4.1.1. First, the path to Blender model of the canvas is acquired by a conjunction of the path to the project folder and the relative path to the root of the project folder and then added to system paths, for example, the model path can be ”`./data/MorseSyrotek/components/Canvas.blend`”. Also, the path to a file where the `Canvas` class is defined is added in the same manner. The `Canvas` class file can be located on the path ”`./src/MorseSyrotek/builder/components/canvas.py`”. The `Canvas` class is imported into the simulation builder-script, and the canvas object is constructed then as indicated in following code fragment.

```
from MorseSyrotek.builder.components import Canvas
canvas = Canvas(filename = canvasPath, alpha = 1.0)
```

The Pen component

The `Pen` component is created as an actuator for a robot in the MORSE simulation as demonstrated in Section 4.1. The actuator provides several groups of functionalities.

The primary functionality is drawing on the canvas, for that the `set_pixel` and the `get_pixel` methods are provided and the `set_line` procedure implementing the Bresenham's algorithm for line drawing [32] is offered. In addition to that, the `get_image_data`, `set_image_data` and `update_image_data` functions which can be used to get or set the whole image with a JSON array serialisation consisting of RGB pixels, or update a set of given pixels with a new RGB value.

The second group of functions provided by the `Pen` component consists of the following methods. The `resize_canvas` procedure changes the resolution of the canvas. The `get_cell_size` function returns the size of one pixel in the map-grid in metres, and the `save_image_copy` function saves the contents of the canvas in a PNG file on a hard-drive.

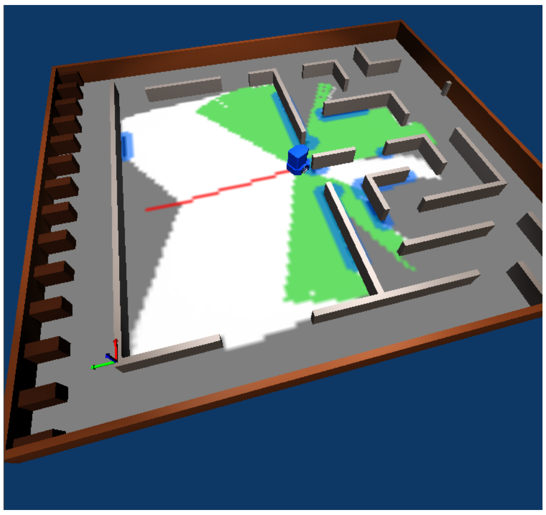
Figure 4.5. On-the-ground canvas mode.

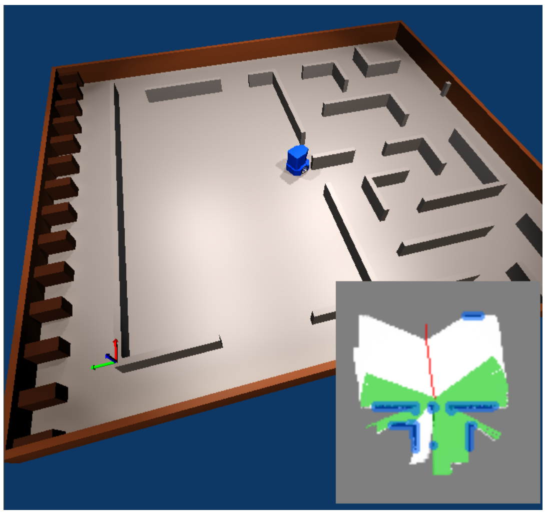
Figure 4.6. Mini-map canvas mode.

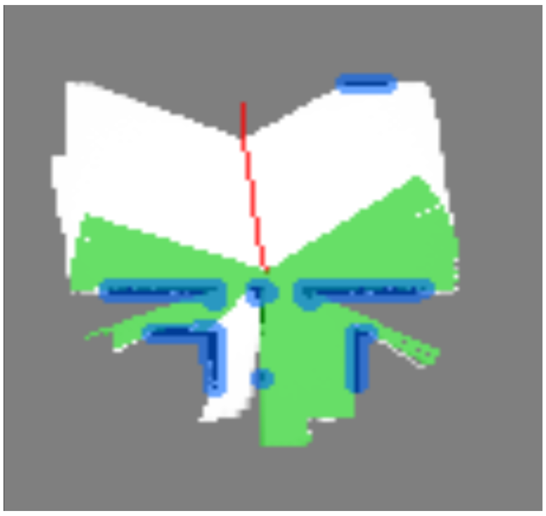
Figure 4.7. Full-screen canvas mode.

The last functionality of the `Pen` class worth mentioning are three modes of canvas visualisation – the canvas-on-the-ground mode (Figure 4.5), the mini-map mode (Figure 4.6) and the full-screen mode (Figure 4.7). The modes can be switched during a simulation by the user with a combination of keys `Left-CTRL + ARROW UP/DOWN`.

The canvas-on-the-ground mode is the standard way of viewing the map-grid, in which the canvas is drawn over the real scene. The user can benefit from this mode because the map-grid can be easily checked whether his/her algorithm uses the map-grid correctly, or the user can display the field of view of the robot's laser rangefinder as the green area demonstrates in Figure 4.5.

The mini-map mode can be used for a quick overview of the scene without the disturbance of the obstacles in the Arena.

The full-screen mode can be utilised for a detailed inspection of the map-grid during the simulation. It can be also configured to be the only way of viewing the map-grid in the simulation (i.e. the mode switching is off). This can be used for a testing of path-planning algorithms, where the user does not need the robot simulation. This scenario will be further described in Section 4.3.2.

4.2.2 Component at the CCMorse side

To control the map-grid component during a simulation, the corresponding class structure is implemented in the CCMorse library. The `MorsePen` class is developed in the second layer of the CCMorse, and it is used to control the `Pen` component mounted on a robot in a simulation.

The `MorsePen` implements the same set of functions as `Pen`, but there are implemented many overloaded functions with various types of parameters for the ease of use. For example, the user can set a pixel with 8-bit RGB values with a specification of alpha (transparency) or without it, and also, the user can set the pixel with a double precision normalized RGB values (0.0–1.0) or with an RGB string code (e.g. "#34AAFF"), this also applies to the line drawing function.

`WindowProxy` class is built on top of `MorsePen` class, and it implements the interface of the Player interface. `WindowProxy` provides a higher-level functionality beside the drawing on the canvas. It can store and manage a multitude of grid-maps. It enables the user to draw several grid-maps on top of each other in layers. The name, colour, transparency of the layer, transparent colour and priority can be set for each layer.

Usage of several grid-maps drawn in layers can be observed in Figures 4.5–4.7, where the green grid cells represent the last scan from the laser rangefinder, the red cells store robot positions and the blue pixels highlight detected obstacles which are inflated so it is certain that the robot does not collide with obstacles when it follows a path planned on the grid-map. All those layers are based on the black-grey-white occupancy grid, where the white means a free explored area, the grey shows the yet unexplored parts of the Arena and the black indicates certain obstacles.

4.3 The Practical robotics course

All features of the CCMorse library and SyRoTek simulation in MORSE, which has been discussed in the preceding text, are used to prepare a set of three complex tasks for students of the Practical Robotics course. The main objective of the course is to increase practical skills of students in robot control and to introduce algorithms for sensor data processing, navigation, map building, planning and intelligent decision making to students.

A base template of a client application is prepared for each task. The template contains a basic structure of the application but the functions implementing robot's behaviour are empty, waiting to be filled by students. Students usually work in pairs and they are encouraged to invent unique solutions for problems introduced in tasks. Students can freely create new classes, functions or other programming constructions in the client code to solve the exercise.

To create the template, the MORSE simulation project has been combined with an existing SyRoTek project template. The combination allows the user to compile their client application for the MORSE simulator or they can compile it for the use on the real SyRoTek system.

Different scenarios are prepared for students with the help of the configurable MORSE simulation builder-script, so the student is able to launch the simulation with two commands – ”import” and ”run”. Importing is done with the `morse import <path> [name]` command, where `<path>` is a path to the root directory of the MORSE simulation and `[name]` is an optional argument that specifies the name of simulation. Example of the simulation import is shown in the following block.

```
$ morse import ~/School/PAR_INTRO
```

The user can run the imported simulation simply by typing the `morse run <name>` command, where the `<name>` is the name of the simulation. The name is identical with the name of the folder where the simulation files are located, unless the user does not declared another name during the import. Example of the use of the run command can be seen in next box.

```
$ morse run PAR_INTRO
```

Tasks for the Practical Robotics course are further described in the following sections.

4.3.1 Introduction task

The introduction task serves, as the name suggests, for the first encounter between students, SyRoTek and MORSE. Students learn the basic concept of robot control and sensor data interpretation.

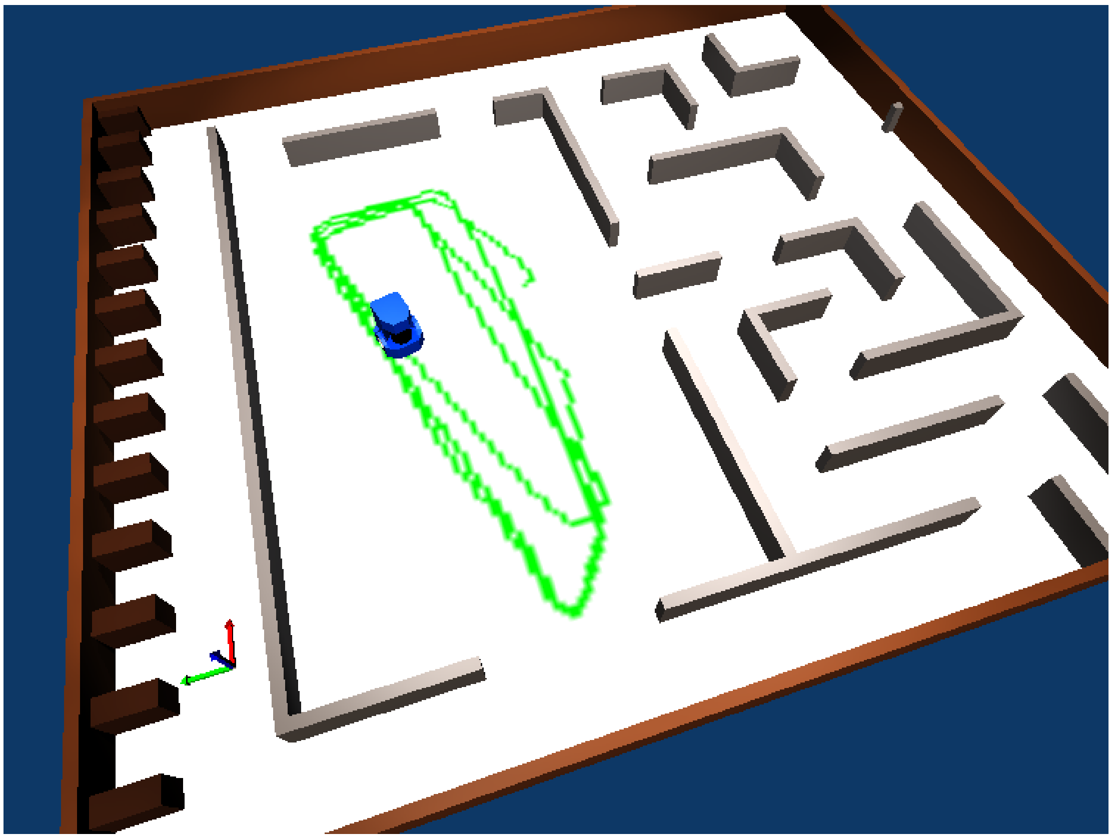

Figure 4.8. Screen-shot of the simulation where the robot performs the simple obstacle avoidance algorithm.

Students have to implement a simple obstacle avoidance algorithm using the laser rangefinder with a behaviour similar to a Braitenberg vehicle, or students can implement the wall-following algorithm. The screen-shot of a simulation with one robot implementing the obstacle avoidance algorithm can be seen in Figure 4.8. The green line shows the robot's motion around the Arena. It is easy to see that the robot just moves on circular path.

4.3.2 Path planning task

The second task of the course is focused on the path planning by using a grid-map. In this task the `Canvas` component of the simulation is configured to be in the full-screen mode without user's ability to change the viewing mode. The simulator serves purely for visualisation of grid-maps in this task.

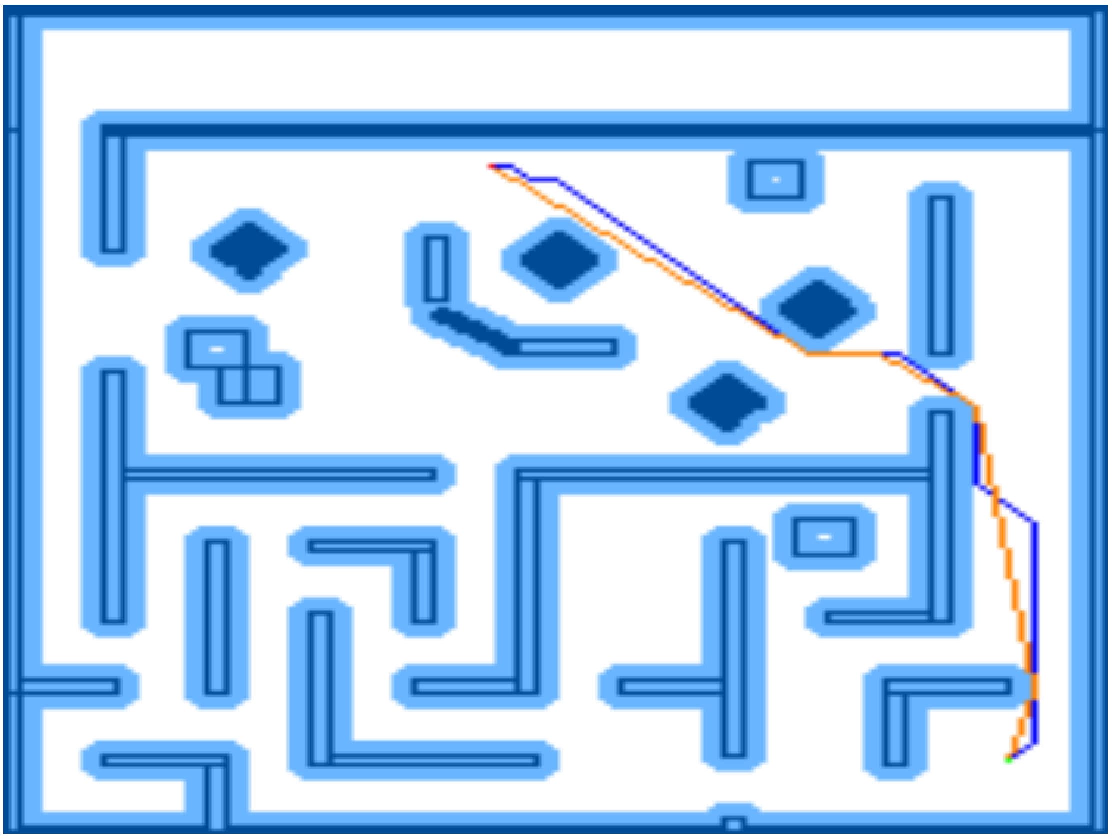

Figure 4.9. Screen-shot of the completed path planning task.

A map of known environment is given to students. Students implement an algorithm for map dilatation, then students interpret the grid-map as an oriented graph. Students use a Dijkstra searching algorithm [33] to find the shortest path from the given start point to the goal. The path found is smoothed in the next step. The results of the path planning can be seen in Figure 4.9, where the light-blue map acts as dilated map, the dark-blue line represents the path found by the Dijkstra's algorithm and the orange line is the smoothed path.

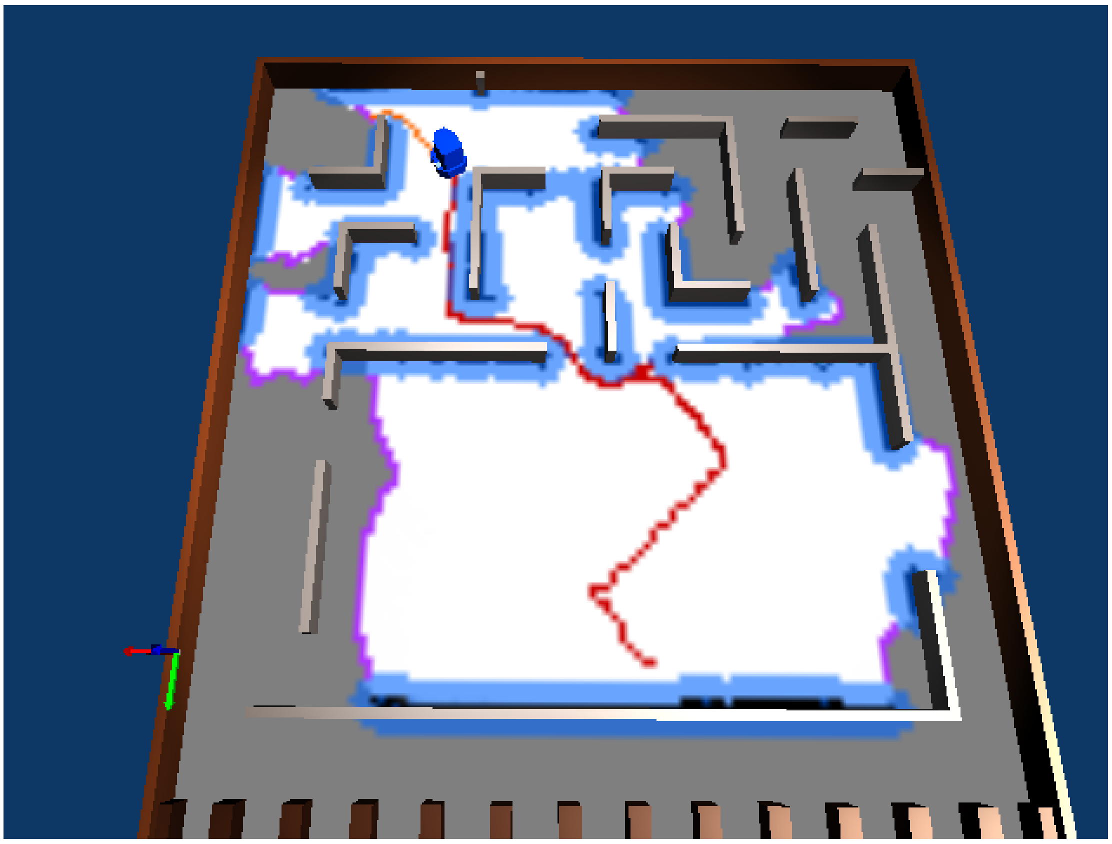

Figure 4.10. Screen-shot of the simulation where the robot performs the exploration of the Arena.

4.3.3 Exploration task

The last task introduces frontier-based exploration of an unknown environment. Students utilise skills from the previous two tasks to control the robot to search the unknown environment and to obtain its map using a multi-threaded client application. One thread of the client is used to collect data and store them in appropriate structures, and the other thread plans the path to the next frontier.

The screen-shot of the simulation where the robot is exploring the Arena is shown in the Figure 4.10. In the figure, the white colour shows the free explored area, and the black represents the certainly found obstacles. Unexplored parts of the Arena are shown in grey. On top of that, the blue overlay indicates the dilated obstacles and the purple shows found frontiers. Furthermore, the orange line represents the planned path to the nearest frontier and the red line is a record of the robot's motion.

4.4 Practical experience with MORSE

I have gained numerous experiences with MORSE during the process of developing this project. While the most of those experiences were pleasant, I have encountered with a few strange behaviours. In the start of the project, the MORSE was used in version 1.2. Then, I have updated to the version 1.4 which is currently the latest stable version of MORSE. Several bugs emerged after the update. Some of the used constructors have new parameters or have slightly different behaviour. These problems were expected and their correction was straightforward with the help of the MORSE's documentation.

But solution of one MORSE's problem has not been found. The MORSE's visualisation of laser scanner's arc (field of view) started to significantly slow down the simulation after the update, to the point where the simulation is virtually unusable. Therefore, the use of this handy feature was abandoned and the laser's arc visualisation is left up to the client application which can animate it using the `Canvas` component.

Chapter 5
Conclusion

The CCMorse library has been successfully developed into a piece of software that enables the user to write a client application in the C++ programming language for control a robot in the MORSE simulation. The CCMorse mimics the back-end application programming interface of the Player middleware, and so the program written using Player interface can also be compiled for a simulation in MORSE. The portability of code has been proven on a few specimens that were originally used for Player/Stage simulations and the SyRoTek system.

The MORSE simulation environment representing the SyRoTek system was combined with the CCMorse library and with SyRoTek's template project to create a set of experiments. These experiments are already being used by students of the Practical Robotics course in the ongoing semester (Winter 2016/2017). After eliminating a few flaws which accompany every deployment of the software to users, the CCMorse library has been working well during the classes. The simple interface of the MORSE simulator and its photo-realistic rendering contributes to a good understanding of robotic problems by students.

The CCMorse library can be easily extended with new devices from the MORSE component library or unique devices developed by the user. Furthermore, the library can be used to implement communication with another interface on top of the Player interface or it can be altered for another simulator then MORSE, because of its layered structure.

5.1 Ideas for future enhancements

Because the CCMorse library is a new software, the development of the library will presumably continue in future. Few suggestions worth mentioning are described in the following paragraphs.

All the time developing the CCMorse library, I have kept an idea for the "Arena controller" on my mind. The arena controller would be a MORSE component allowing to control the dynamic obstacles in the SyRoTek's Arena. The controller would be configured with a set of predefined obstacle positions, timers for actions of obstacles and traps based on the position of a robot in a simulation. This functionality will bring more control over the simulation and the user would be able to simulate more complex experiments.

Another thought to be considered is to implement a different communication protocol between the MORSE simulator and the CCMorse library or to find more efficient way of using the Socket. Although the Socket communication is sufficient for majority of single robot experiments, it appears to be a bit time consuming when transmitting a larger amount of data, e.g. an update of data in the whole canvas. Larger sets of data must be sent over multiple messages, which slows down the execution of a client application.

The last idea suggests a creation of a mode where the client application would run simultaneously on both the simulation and the real system. The simulator would be used as a visualizer of the data collected by a robot in the real environment.

References

[1] Kulich, M., J. Chudoba, K. Kosnar, T. Krajnik, J. Faigl, and L. Preucil. SyRoTek—Distance Teaching of Mobile Robotics. *IEEE Transactions on Education*. feb, 2013, Vol. 56, No. 1, pp. 18–23. ISSN 0018-9359. Available from DOI 10.1109/TE.2012.2224867.

[2] *SyRoTek Web* [online].
`https://syrotek.felk.cvut.cz/`.

[3]

[4] Bakken, David. Middleware. In: *Encyclopedia of Distributed Computing*. Dodrecht, Netherlands: Kluwer Academic Publishers, 2001.
`http://www.eecs.wsu.edu/~bakken/middleware.pdf`.

[5] Elkady, Ayssam, and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*. Hindawi Publishing Corporation, jan, 2012, Vol. 2012, pp. 15. Available from DOI 10.1155/2012/959013.

[6] Gerkey, B. P., R. T. Vaughan, K. Støy, A. Howard, G. S. Sukhatme, and M. J. Mataric. Most Valuable Player: A Robot Device Server for Distributed Control. In: *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*. Wailea, Hawaii, 2001. pp. 1226–1231. Available from DOI 10.1109/IROS.2001.977150.

[7] Gerkey, Brian P., Richard T. Vaughan, and Andrew Howard. ”The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems”. In: *In Proceedings of the 11th International Conference on Advanced Robotics*. University of Coimbra, Portugal, 2003. pp. 317–323.
`http://robotics.usc.edu/publications/288/`.

[8] *Player Manual* [online].
`http://playerstage.sourceforge.net/doc/Player-3.0.2/player/`.

[9] *The Player/Stage Web* [online].
`http://playerstage.sourceforge.net/`.

[10] Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In: *Proceedings of the Workshop on Open Source Software (ICRA)*. 2009.
`http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf`.

[11] *ROS: Robot Operating System Web* [online].
`http://www.ros.org`.

[12] Koenig, N., and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In: *2004 IEEE/RSJ International Conference*

on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566). 2004. pp. 2149–2154. Available from DOI 10.1109/IROS.2004.1389727.

[13] *Open Dynamics Engine* [online].
`http://www.ode.org`.

[14] *Bullet: Real-Time Physics Simulation* [online].
`http://www.bulletphysics.org/`.

[15] *Blender 3D* [online].
`http://www.blender.org/`.

[16] *Simbody: Multibody Physics API* [online].
`http://simtk.org/projects/simbody`.

[17] *DART: Dynamic Animation and Robotics Toolkit* [online].
`http://dartsim.github.io`.

[18] Michel, Olivier. Cyberbotics Ltd. Webots: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems*. dec, 2004, Vol. 1, No. 1, pp. 39–42. Available from DOI 10.5772/5618.
`http://arx.sagepub.com/content/1/1/5.abstract`.

[19] Rohmer, E., S. P. N. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013. pp. 1321–1326. ISSN 2153-0858. Available from DOI 10.1109/IROS.2013.6696520.

[20] *Vortex Dynamics* [online].
`http://www.vxsim.com/`.

[21] Echeverria, G., N. Lassabe, A. Degroote, and S. Lemaignan. Modular OpenRobots Simulation Engine: MORSE. In: *2011 IEEE International Conference on Robotics and Automation*. 2011. pp. 46–51. ISSN 1050-4729. Available from DOI 10.1109/ICRA.2011.5980252.

[22] Echeverria, Gilberto, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. Simulating Complex Robotic Scenarios with MORSE. In: Itsuki Noda, Noriaki Ando, Davide Brugali, and James J. Kuffner, eds. *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 2012. pp. 197–208. SIMPAR'12. ISBN 978-3-642-34326-1. Available from DOI 10.1007/978-3-642-34327-8_20.

[23] *The MORSE Simulator Documentation* [online].
`https://www.openrobots.org/morse/doc/stable/morse.html`.

[24] *YARP Documentation* [online].
`http://www.yarp.it/`.

[25] *Pocolibs* [online].
`http://pocolibs.openrobots.org`.

[26] *MOOS Documentation* [online].
`http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/`.

[27] *MAVLink: Micro Air Vehicle Communication Protocol* [online].
`http://qgroundcontrol.org/mavlink/start`.

[28] *JSON: JavaScript Object Notation* [online].
`http://www.json.org/`.

[29] *Boost C++ Libraries* [online].
`http://www.boost.org`.

[30] *Blender Tutorials* [online].
`https://www.blender.org/support/tutorials/`.

[31] *Blender Manual* [online].
`https://www.blender.org/manual/`.

[32] *Bresenham's line algorithm* [online].
`https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm`.

[33] *Dijkstra's algorithm* [online].
`https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`.

Appendix A
Specification

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Bertl Lukáš**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Aplikační rozhraní pro simulátor MORSE**

Pokyny pro vypracování:

1. Seznamte se se systémem Player/Stage (http://playerstage.sourceforge.net), simulátorem MORSE (https://www.openrobots.org/wiki/morse/) a modelovacím nástrojem Blender (https://www.blender.org).
2. Seznamte se s výukovým systémem SyRoTek (https://syrotek.felk.cvut.cz) a úlohami řešenými v kurzu Praktická robotika.
3. Vytvořte model robotu S1R a Arény SyRoTek pro simulátor MORSE.
4. Navrhněte a implementujte aplikační rozhraní v C++ pro ovládání robotu v simulátoru MORSE.
5. Nad implementovaným rozhraním realizujte další rozhraní, které plně nahradí rozhraní Playeru používané v kurzu Praktická robotika.
6. Kód vyzkoušejte jak v simulátoru MORSE, tak s reálným robotem systému SyRoTek.
7. Implementovaná rozhraní důkladně zdokumentujte.

Seznam odborné literatury:

[1] Kulich, M.; Chudoba, J.; Košnar, K.; Krajník, T.; Faigl, J.; Přeucil, L., "SyRoTek-Distance Teaching of Mobile Robotics," Education, IEEE Transactions on , vol.56, no.1, pp.18,23, Feb. 2013

[2] A. Kelly: Mobile Robotics: Mathematics Models and Methods, Cambridge University Press, 2014, ISBN: 9781107031159

Vedoucí: RNDr. Miroslav Kulich, Ph.D.

Platnost zadání: do konce letního semestru 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	děkan

V Praze dne 30. 11. 2016

Appendix B
The contents of the CD

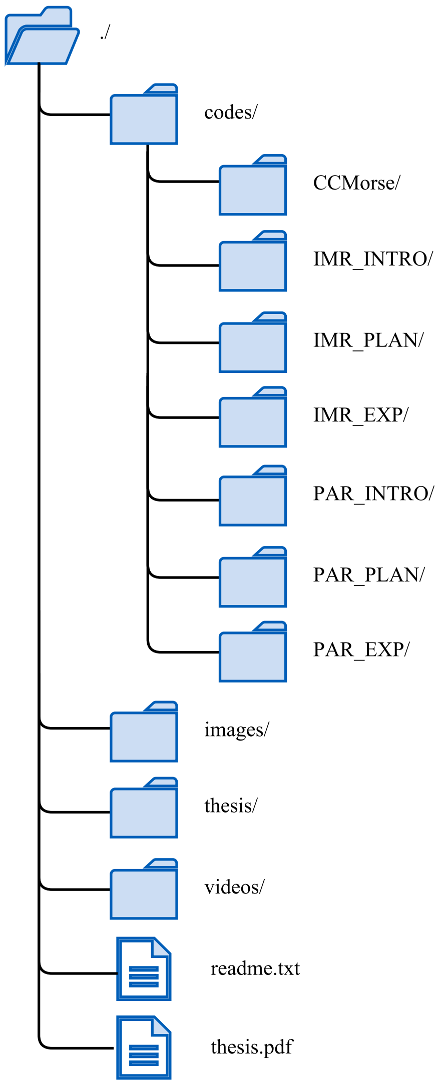

Figure B.1. Diagram showing the folder structure of the CD.

The Figure B.1 shows the contents of the enclosed CD. The thesis is in the file named `thesis.pdf` and source files of the thesis are stored in the `thesis` directory. The `codes` directory contains the source files of the CCMorse library and two implementations of each task from the Practical robotics course. Directories `images` and `videos` contain additional media files.

Appendix C
Code listings

C.1 robot.h

```cpp
#ifndef __ROBOT_H__
#define __ROBOT_H__

#ifdef _MORSE
    #include "libCCMorse/CCMorse.h"
#else
    #include "libPlayer/Position2dProxy.h"
    #include "libPlayer/LaserProxy.h"
    #include "libPlayer/PlayerClient.h"
    #define PI 3.14159265359
#endif

#include "imr-h/imr_config.h"
#include "imr-h/thread.h"
#include "robot_types.h"

namespace imr { namespace robot {

class CRobot : public imr::concurrent::CThread {
    typedef imr::concurrent::CThread ThreadBase;

    public:
        static imr::CConfig& getConfig(imr::CConfig& config);
        CRobot(imr::CConfig& cfg);
        ~CRobot();
        void stop(void);

    protected:
        void threadBody(void);
        void navigation(void);

    private:
        imr::CConfig& cfg;
        CCMorse::PlayerClient *client;
        bool quit;
        bool alive;
        SPosition pos; // current robot position
        CCMorse::LaserProxy *laser;
        CCMorse::Position2dProxy *position2d;
        Mutex mtx;

}; // class CRobot
} } // namespace imr::robot
#endif
```

C.2 robot.cc

```
#include <unistd.h> //usleep
#include <fstream>

#include "imr-h/logging.h"
#include "imr-h/imr_exceptions.h"
#include "robot_types.h"
#include "robot.h"
#include "libCCMorse/MorseError.h"

using namespace imr::robot;
using namespace CCMorse;

typedef unsigned char uchar;


imr::CConfig& CRobot::getConfig(imr::CConfig& config)
{
#ifdef _MORSE
    config.add<std::string>(
        "host", "morse server hostname", "localhost"
    );
    config.add<int>("port", "morse server port", 4000);
#else
    config.add<std::string>(
        "host", "syrotek server hostname", "syrotek.felk.cvut.cz"
    );
    config.add<int>("port", "syrotek server port", 38000);
#endif // _MORSE
    config.add<int>("laser-index", "laser index", 0);
    config.add<int>("position2d-index", "position2d index", 0);
    config.add<bool>(
        "coords",
        "enable/disable settings of initial coords of the robot",
        false
    );
    config.add<double>("x", "initial x position of the robot", 0.0);
    config.add<double>("y", "initial y position of the robot", 0.0);
    config.add<double>("yaw", "initial yaw of the robot ", 0.0);
    return config;
}

CRobot::CRobot(imr::CConfig& cfg) : ThreadBase(), cfg(cfg), client(0),
    quit(false), laser(0), position2d(0)
{
    client     = new CCMorse::PlayerClient(
        cfg.get<std::string>("host"), cfg.get<int>("port")
    );
    laser      = new LaserProxy(
        client, cfg.get<int>("laser-index")
    );
    position2d = new Position2dProxy(
        client, cfg.get<int>("position2d-index")
    );

#ifdef _MORSE
    CCMorse::WindowProxy::GetInstance(client, 172, 186);
#endif // _MORSE

    assert_argument(laser, "Can not create laser proxy");
    assert_argument(position2d, "Can not create position2d proxy");

    // check laser
    while (!laser->IsFresh()){  // laser init retry
        client->Read();
        while(client->Peek(0)) {
            client->Read();
        }
    }

    DEBUG("CRobot: Done.");
}

CRobot::~CRobot()
{
    stop();
    join();

    if (client) {
#ifdef _MORSE
        delete &CCMorse::WindowProxy::GetInstance();
#endif // _MORSE
        delete laser;
        delete position2d;
        delete client;
    }
}

void CRobot::stop(void)
{
    ScopedLock lk(mtx);
    quit = true;
}

void CRobot::threadBody(void)
{
    try {
        navigation();

    } catch (CCMorse::MorseError& e) {
        ERROR(
            "Morse error: " << e.GetErrorStr() << " function: "
            << e.GetErrorFun()
        );

    } catch (imr::exception& e) {
        ERROR("Imr error: " << e.what());
        exit(-1);
    }
}

void CRobot::navigation(void)
{
    bool q = false, laserFirst = true;
#ifdef _MORSE
    WindowProxy& window = CCMorse::WindowProxy::GetInstance();
    ImagePoint_t last, current;
    bool poseFirst = true;
#endif // _MORSE

/// user variables ---------------------------------------
    double const VMAX = 0.3, WMAX = 1.0, RNG_ANGL_REACTION = 0.5,
                 HALF_FOV = (40 * (PI/180)), MIN_OBSTACLE_RANGE = 0.20,
                 SAFE_OBSTACLE_RANGE = 0.50;

    unsigned LASER_INDEX_FROM_CENTER_TO_EDGE = 340,
             LASER_CENTER_INDEX = 340, LASER_LEFT_INDEX = 0,
             LASER_RIGHT_INDEX = 681;

    double v = 0.0, w = 0.0, MAX_RANGE = 5.0, lMin = 5.0, rMin = 5.0,
           range = 5.0, lMinAngle = 0.0, rMinAngle = 0.0, angle = PI;

    int direction = 1;
/// end of user variables ---------------------------------

    DEBUG("Navigation started.");

    position2d->SetMotorEnable(true);

    do {
        client->Read();
        while(client->Peek(0)) {
            client->Read();
        }


        if (position2d->IsFresh() && position2d->IsValid()) {
            // Get and save current robot position.
            pos.x = position2d->GetXPos();
            pos.y = position2d->GetYPos();
            pos.yaw = position2d->GetYaw();
            // Log robot position.
            DEBUG(
                "POSITION  " << pos.x << " " << pos.y << " " << pos.yaw
            );
            DEBUG(
                "SPEED  " << position2d->GetXSpeed() << " "
                << position2d->GetYSpeed() << " "
                << position2d->GetYawSpeed()
            );
            // First init
#ifdef _MORSE
            if(poseFirst){
                poseFirst = false;
                last = ImagePoint_t (
                    window.GetX(pos.x), window.GetY(pos.y)
                );
                current = last;
            }
            // Drawing robot trajectory for last cycle.
            current = ImagePoint_t (
                window.GetX(pos.x), window.GetY(pos.y)
            );
            window.SetLine(
                last.x, last.y, current.x, current.y, 0, 255, 0
            );
            window.SetPixel(current.x, current.y, 255, 0, 255);
            last = current;
#endif // _MORSE
            // Flaging this data as not fresh.
            position2d->NotFresh();
        }

        if (laser->IsFresh() && laser->IsValid()) {
            // Log laser data.
            DEBUG(
                "laser " << laser->GetMinAngle() << " "
                << laser->GetMaxAngle() << " " << laser->GetCount()
            );
            // First init
            if(laserFirst){
                MAX_RANGE = laser->GetMaxRange();
                LASER_INDEX_FROM_CENTER_TO_EDGE = (unsigned)
                    (HALF_FOV / laser->GetScanRes());
                LASER_CENTER_INDEX = (unsigned) (laser->GetCount() / 2);
                LASER_LEFT_INDEX   = (LASER_CENTER_INDEX -
                    LASER_INDEX_FROM_CENTER_TO_EDGE);
                LASER_RIGHT_INDEX  = (LASER_CENTER_INDEX +
                    LASER_INDEX_FROM_CENTER_TO_EDGE);
                lMin = MAX_RANGE, rMin = MAX_RANGE, range = MAX_RANGE;
                laserFirst = false;
            }
            // Calculate new velocities from new laser data
            lMin = MAX_RANGE;
            rMin = MAX_RANGE;
            unsigned lId = 0;
            unsigned rId = laser->GetCount();

            for(
                unsigned i = LASER_LEFT_INDEX;
                i < LASER_CENTER_INDEX;
                i++
            ){
                if( (*laser)[i] < lMin ){
                    lMin = (*laser)[i];
                    lMinAngle = laser->GetBearing(i);
                    lId = i;
                }
            }
            for(
                unsigned i = LASER_RIGHT_INDEX;
                i > LASER_CENTER_INDEX;
                i--
            ){
                if( (*laser)[i] < rMin ){
                    rMin = (*laser)[i];
                    rMinAngle = laser->GetBearing(i);
                    rId = i;
                }
            }

            if(lMin < rMin){
                range = lMin;
                angle = lMinAngle;
                direction = 1;

            } else {
                range = rMin;
                angle = rMinAngle;
                direction = -1;
            }

            if(range > MIN_OBSTACLE_RANGE){
                if(range < SAFE_OBSTACLE_RANGE){
                    v = ((range - MIN_OBSTACLE_RANGE)/(
                        SAFE_OBSTACLE_RANGE-MIN_OBSTACLE_RANGE
                        )) * VMAX;
                    w = ( (
                            ( RNG_ANGL_REACTION * (
                                MIN_OBSTACLE_RANGE / range
                            ) )
                            +
                            ( (1-RNG_ANGL_REACTION) * (
                                HALF_FOV*2 - angle /HALF_FOV*2
                            ) )
                        ) * WMAX * direction );

                } else {
                        v = VMAX;
                        w = 0.0;
                    }
                } else {
                    v = 0.0;
                    w = WMAX;
                }

                // Flaging this data as not fresh.
                laser->NotFresh();
            }

/// command the robot -------------------------------------

            position2d->SetSpeed(v, w);

/// end of commands ---------------------------------------

            ScopedLock lk(mtx); // check quit request
            q |= quit;
        } while (!q);

        position2d->SetMotorEnable(false);

        INFO("Main loop has been left");
}
```