AG ROBOTERSYSTEME
FACHBEREICH INFORMATIK
AN DER UNIVERSITÄT KAISERSLAUTERN

Projektarbeit

SubSim - An Autonomous Underwater Vehicle Simulation System

Tobias Bielohlawek

April 27, 2006

SubSim - An Autonomous Underwater Vehicle Simulation System

Projektarbeit

Arbeitsgruppe Robotersysteme
Fachbereich Informatik
Universität Kaiserslautern

Tobias Bielohlawek

Tag der Ausgabe : 10. Oktober 2005
Tag der Abgabe : April 27, 2006

Betreuer : Prof. Dr. Thomas Bräunl
Referent : Prof. Dr. Karsten Berns

Ich erkläre hiermit, die vorliegende Projektarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den April 27, 2006

(Tobias Bielohlawek)

Contents

1. Introduction

"A simulation is an imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system."[SIM 05]

1.1 Motivation

The CIIPS[1] group at the University of Western Australia in Perth is active in developing submarine robots, so called "Autonomous Underwater Vehicles", or AUV. Those models are controlled by an embedded microcontroller running an individual user program. Developing, debuging and testing those controller programs on physical submarine models is quite time consuming and circuitous. A simulation system avoids the need for a physical model and an underwater environment. This results not only in time savings but also decreasing development costs.

1.2 Project Aim

The aim of this software engineering project is to design and develop a simulation program. It is meant as a debug and testing system for researchers, developing programs for submarine robot controller.

The focus is especially set on a system with a dynamic physical simulation which executes a controller program, simulates and visualizes sensor data, actuator behavior and camera pictures.

The basic program structure with its several program parts and the interfaces in between them is to be defined. There are parts for the physical simulation of the AUV model, three dimensional representation of the environment or the graphical user interface. For the physical simulation, the CIIPS group provides the library PAL, which has to be included.

To execute and simulate the controller program the thesis introduces a basic, universal interface which links a controller program with the simulation system. As the

[1] CIIPS: Centre for Intelligent Information Processing Systems - http://ciips.ee.uwa.edu.au

CIIPS group uses the EyeBot controller for the submarine models, a special interface for this controller is to be designed and implemented as well.

To extend the system in the future, the focus is additionally set on a structure allowing to include plugins easily.

1.3 Project Structure

Chapter 2 describes the initial situation with an overview of the submarine projects at the CIIPS group. A discussion of other simulation systems follows which leads to the program requirements for the new system at the end

Chapter 3 introduces the software concept and discusses first program components necessary for the realization and implementation of the system.

Chapter 4 describes the program components in more detail. The system is split in a server part with a layer model and client part. The server is presented with a focus on the data layer which keeps the data structure and physical simulation part. The different types of clients connecting to the server are explained with the introduction of the Eyebot Plugin as an example plugin and implementation of the EyeBot controller interface.

An example simulation case is given in chapter 5. It is meant to demonstrate the program usage and its features.

Chapter 6 finishes the thesis with a final summary and gives outlook to future developments and improvements.

The appendix features an overview of the interfaces (A), example settings files (B), class hierarchy (C), a short manual (D) and GUI shortcuts (E).

2. Initial Situation

2.1 Autonomous Underwater Vehicles

An AUV model typically consists of an embedded controller with a set of hardware components like actuators, sensors and a camera. Typical actuators are electronic motors or servos, example sensors are PSD (Positional Sensitive Device) for distance measuring or contact sensors. All the devices are connected to the controller running a special program to control the system.

There are two different AUV developments running:

The first AUV developed at the CIIPS group is named *MAKO*. The aim of this project is to gain basic knowledge about developing AUVs. Besides constructing the model and embedding the EyeBot controller, the focus is also set on sensor development. The AUV is slightly buoyant and has an automatic righting effect because of its low center of mass.

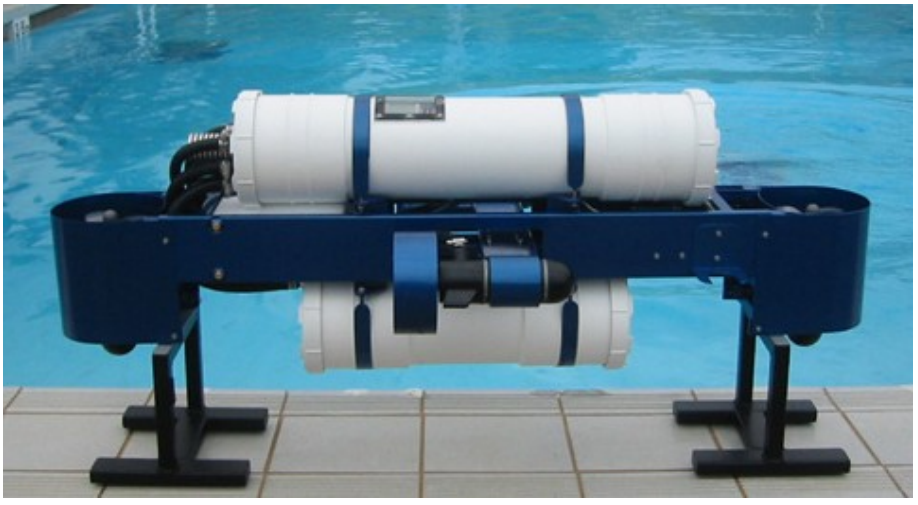

Figure 2.1: Autonomous Underwater Vehicle MAKO

An EyeBot controller, connected to four propeller motors, a velocimeter and four echo sounders controls the submarine. The motors act independent of each other,

two for moving and two for active diving. An onboard color camera is used for obstacle recognition to follow and detect objects, for example. Bluetooth, WLAN and an infrared remote control are used for communicating to a host station once the AUV has surfaced.

The second project is about a submarine model called *USAL*. Compared to the AUV *MAKO*, it is much smaller and more compact, therefore handier to transport. It is constructed to provide a basement for future research and to participate in AUV competitions.

Figure 2.2: Autonomous Underwater Vehicle USAL

The AUV is equipped with two thrusters, one for movement forward/backward (x-direction) and one for movement in up/down (z-direction). A rudder blade in the back provides yaw movement when the AUV is going in x-direction. For providing yaw movement in small operation areas, an independent small bow rudder is additionally attached. The model is controlled by an EyeBot controller with an interface to a color camera. A huge sensor suite for mission and navigation tasks is integrated: distance measurement to the front, the sides and bottom direction, a compass, a 3-axis accelerometer and a depth sensor. To prevent the AUV from damage in the case of leakage, an inside water sensor is integrated. Communication is provided via Bluetooth for up- and downloading data and basic instructions can be send by a infrared remote control.

2.2 Other Simulation Systems

As one can imagine, developing (EyeBot) controller programs and testing them on physical AUVs is quite time consuming and circuitous. The special underwater environment leads to a completely new situation compared to known environments like those for robots running on floors. First, it is obvious that a water pool is needed which is uncommon in usual research places. Second, special precautions have to be taken for an underwater recovery of the model in case of a program or hardware failure. Still, the risk to gain hardware damage by water is quite high which results in increasing development costs.

The AUV models are kept compact, hence the amount of included batteries is restricted. External power supply with cables is through the risk of a circuit shortcut caused by water hardly possible. This leads to a certain amount of time testing an EyeBot program on the AUV before having to recharge or change the batteries. For testing purposes, onboard data tracking is a common technique, but as of the limited space in the AUV model and therefore missing storage devices this is hardly possible.

Common AUV applications like avoiding or following obstacles need special underwater environment with different kind of objects. Those objects must be controlled in a real water pool, which can be quite complicated, considering changing position or performing special movements.

Additionally, a simulation system plays an important role, since it allows to develop and test the controller program right from the beginning. Is the physical robot model finished, the program can be transferred to the controller without modifications. The real-time simulation system can help closing the gap between off-line simulation and real testing using the already implemented robot model. When properly interfaced with the robot hardware, a real-time graphical simulation with a "hardware in the loop" (HIL) configuration, allows running the implemented controller program on the actual robot model. This reduces the development time drastically.

The following introduces two simulation systems motivated by this scenario. The first one is *Neptune* developed for research purposes, the second is *DeepWorks*, which is targeted to industry field and commercial applications.

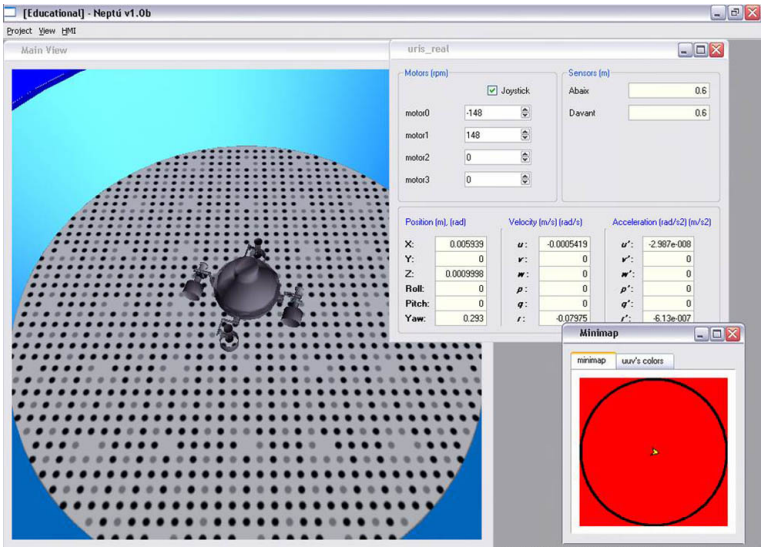

Figure 2.3: Neptune screenshot showing the graphical represenation of simulation and sensor data.

The system *Neptune* is a real-time 3D graphical simulator for running online, hardware in the loop and hybrid simulations. It is developed by the Computer Vision and Robotics Group at the University of Girona in Spain[1] and first introduced in 2002.

[1] VicoRob - http://vicorob.udg.es

The system is very flexible in the sense that new virtual worlds and new AUV models can be added in a very easy way. As a multi-vehicle Simulator more than one robot can be simultaneously simulated. Each robot is defined through three basic files with geometry, hydrodynamics and sensors definition. Simulated sensors for range detection, sonar, vision and internal sensors (position, attitude, speed and depth) are supported. The environment is modeled using VRML combined with a bathymetry model. The graphical simulator is based on OpenGL. A screenshot of the program is shown in figure 2.3.

In order to allow a real time performance, the application has been build as a distributed application including several processes: the *Neptune* main program, one robot dynamics process for each simulated robot and a name server. All the programs interact among them through a TCP/IP network.

Ocean currents and waves are no yet supported, but it is planned to introduce them in next versions. A more detailed description can be found in [Ridao 04].

Another interesting simulation system is *DeepWorks* - a high-end world class subsea simulation. It is a suite of simulation solutions developed by GLR[2] situated in England. Founded in 1988, GRL is the one of world leader at developing simulation solutions for the subsea environment.

DeepWork has a unique software architecture for the 3D simulation, visualisation and control of underwater systems. The System is developed for the Offshore industry who rightly demands the highest levels of quality and reliability. The simulation system consits of core libraries which provide:

- User-configurable 3D graphical models
- Contact dynamics to detect collisions and respond correctly.
- Manipulator and vehicle control
- Vehicle/vessel dynamics and hydrodynamics (ROV?s, submarines)
- Environmental factors (lighting, clouding, turbidity)
- Cable dynamics and hydrodynamics
- Graphics engine with network support
- Currents and tides (3D and time varying)
- Sonar simulation based on line tracing
- Real time interfacing to position sensing data (NMEA for example)

The libraries are available for customers to incorporate into their own products.

[2]GLR - http://www.generalrobotics.co.uk

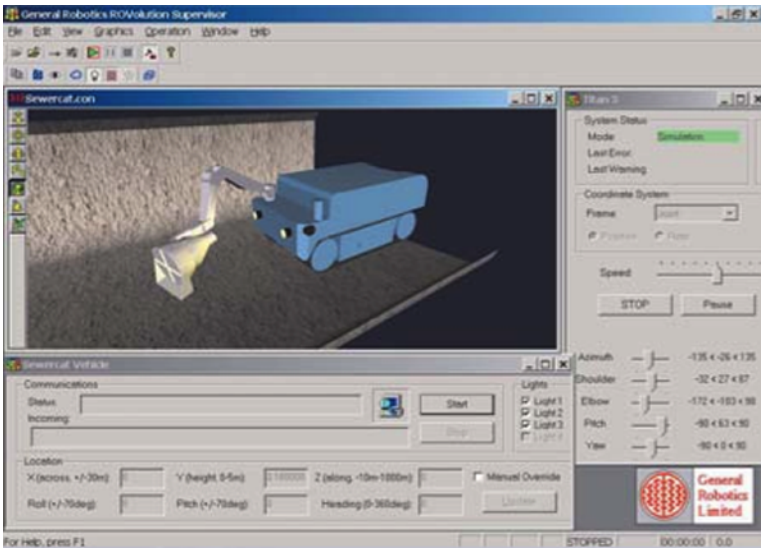

Figure 2.4: DeepWorks screenshot showing a running simulation and the control panels.

2.3 Requirements

The examples show, that powerful underwater simulation systems are already available. Plenty of different applications can be simulated. But nevertheless the simulated robot models and embedded controllers have to be tailored and configured to fulfill certain requirements. A gerneral, universal implementation is yet not supported. A customization to the EyeBot controller used by CIIPS group is not possible.

This leads to the decision to design and implement a new simulation system with and universal interface for generic controller systems and an interface implementation for the EyeBot controller. The program is named ”SubSim”, a merge of the words ”Submarine” and ”Simulator”, clearly hinting this project’s aim.

Based on these facts, the following requirements are made:

- The underwater environment with its special physical circumstances is be simulated together with all the different sensor inputs including the cameras of an AUV.
- For special scenarios, other additional passive objects like a floating buoy, ship or pipes should be placeable in the environment to interfere with the AUV.
- Any EyeBot program written for an AUV should be allowed to execute and to be simulate without restrictions or changing the source code.
- As a simulation without being able to monitor its progress is quite useless, the system should enable the user to watch the environment in different views and observe all the sensor data from a chosen AUV while simulating.
- The movements of the objects are to be visualized as well the controller output to its LCD display.

- The user must have the opportunity to interact by changing simulation speed or object position.
- To extend the simulator for future applications a plugin system should be implemented which offers access to the functions and data of the simulator program through a special interface.
- As it is common practice, any settings file for the application or describing the objects should be structured in XML (Extensible Markup Language) to provide an intuitive and easy editing.
- Endusers should get support through a help system
- Developer documentation for creating plugins and continuing the work on the simulation program development should be provided.

This thesis comes out with a discussion of a simulation system fulfilling these requirements. It shows how the requirements are implemented and realized.

3. Concept of the Simulation System

A usual simulation system consists of several parts interacting with each other. As shown in figure 3.1 the simulation core is in the center of the system. It executes a controller program and process the physical calculations according to the information supplied by the data part. The data part contains the graphical and physical model of the world, robots and other objects. To be highly customizable, the simulation and model descriptions are provided through user-editable setting files.

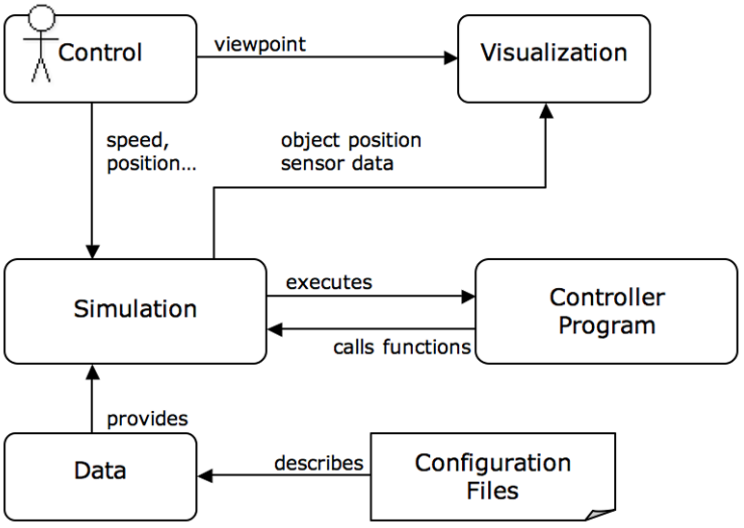

Figure 3.1: Concept of the Simualtion System

The simulation results, as output, are presented in the visualization part. In here, simulation and sensor data is displayed and a three-dimensional model of the environment is rendered. The control part allows the user to input and interact with the simulation. The user is able to control the simulation in changing the execution speed or adjusting parameters. The users viewpoint in the rendered scene and visualization mode of the objects can be changed as well.

3.1 Definitions

In this document the underwater environment will be referred as *World*. The three-dimensional objects acting in the environment are called *WorldObjects*. There are active and passive objects. Passive examples are buoys, pipelines or any other object the submarine may interact with. An active object is an AUV (e.g. submarine) and called *WorldActiveObject*. Compared to a passive object it additionally contains a proper description of its sensors, actuators and cameras together with an (EyeBot) controller program file called *Client* in the following. To detect and use the hardware components, the EyeBot controller uses a mechanism called the *Hardware Description Table*, or *HDT*, for defining low-level attributes for sensors and actuators. Typical actuators are electronic motors or servos, example sensors are PSDs (Positional Sensitive Devices) for distance measuring or contact sensors for detecting touch impulses. For image recognition, the EyeBot controller supports a monochrome or a color camera.

In order to accurately simulate the underwater environment, a proper physics simulation is needed. The AUV, as an active object, and additional passive objects are supposed to interact with the environment in a similar manner to the physical reactions of real objects. The influences of forces, collisions and as well liquid effects have to be considered. Virtual sensors like a PSD or camera have to act like their real counterparts, so the controller program deals with the same data as in the real environment.

The 3rd party library "Physics Abstraction Layer" (PAL) [PAL 05] with the physics engine "Newton Dynamics"[1] offers simulating an underwater environment. PAL wraps an abstraction layer around the engine and allows additional implementations for simulating hardware devices. As a free and fast open source library supporting water scenarios, Newton was the physics engine of choice.

For visualizing the *World* and its *WorldObjects* on the computer screen, a three dimensional representation of the graphical objects is needed. The common program and popular file format, "Milkshape 3D " (MS3D)[2] is used for graphical model files. The low polygon 3D modelling program is freely distributed and widely spreaded. It easily allows creating complex graphical models out of primitive geometries, which can be exported and displayed in 3D rendered scenes.

OpenGL (Open Graphics Library)[3] realizes the drawing of the objects. It offers 3D model rendering on the computer screen from any viewpoint of the observer. It is a specification defining a cross-language and cross-platform API. The interface consists of over 250 different function calls that can be used to draw complex three-dimensional scenes from simple primitives. OpenGL is widely spread in video game development, CAD (computer aided design) and scientific visualization. A good introduction to OpenGL can be found in [Dave Shreiner 05]

With this technique, SubSim can draw any type of obstacle that is constructed from a set of primitive geometries. The obstacle will be simulated using the physical representation, similar to the process of physically modeling the submarine. Obstacles

[1] Newton Dynamics - http://www.newtondynamics.com
[2] Milkshape 3D - http://www.swissquake.ch/chumbalum-soft
[3] OpenGL - http://wwww.opengl.org

can be constructed at any position and orientation with any desired mass. This even allows obstacles to float in the water, if they are given a low density value.

The controller program running on the EyeBot controller of an AUV is coded in the programming language C, using its standard library function. For interacting with the controller and its connected devices, special EyeBot functions are added.

All those special functions form together the application-programming interface (API) called RoBIOS. The word RoBIOS stands short for Robot BIOS or Robot basic input/output system. The functions are divided in two different parts: the low-level and the high-level API. Whereas the low-level API provides universal functions for basic in/output processing, the high-level API forms an abstraction layer by extending the low-level API for functions interacting easily with certain actuators, sensors and cameras. To know which hardware device is connected to which port of the controller, a *HDT* (*Hardware Description Table*) is used, defining low-level attributes for the hardware.

To achieve the goal to simulate and execute AUV controller programs, it is necessary to provide at least an implementation of the low-level API together with the *HDT*. The EyeBot high-level API can be sourced out in an external part of the simulation system, in a so-called EyeBot plugin. SubSim uses plugins for extending the program core and adding new features. A plugin has to be provided as dynamic linked library (DLL) and is loaded at program startup or during runtime by a plugin manager. The high-level API, realized as a *SubSim Plugin*, is flexible to different RoBIOS versions and even to other controller implementations. This is clearly an advantage: does the high-level API change, only the EyeBot plugin has to be re-adjusted, not the SubSim core program and its low-level API.

As the EyeBot controller is used at the CIIPS group at the University of Western Australia in Perth, the focus is especially set on this controller.

The controller is developed as a very versatile embedded robot controller system, suitable to build lots of different applications in the context of autonomous mobile robots, like small 2-wheel robots, omni directional robots, multiple leg robots or humanoids. An overview of the capabilities and the different applications of the EyeBot controller is presented in[Thomas Bräunl 00] and [Bräunl 01].

3.2 Software Architecture

The simulation system ”SubSim” uses a client-server architecture. The server consists of the program core, the data processing unit and the Graphical User Interface (GUI). Interfaces to the so called ”Clients”, which is either a Plugin or a controller program, are provided through the two APIs ”APIPlugin” and ”APILow-Level”.

Figure 3.2 gives an overview of the software architecture. The name of the main class in each layer is printed in italics.

The server itself is structured in a multi layer based model divided in a presentation-, control-, application- and data-layer. Besides the application settings and the class *World* keeping the simulation data, the data-layer additionally contains two modules: the ”Physics Module” for processing the physical simulation and the ”Graphics Module” for handling the object visualization.

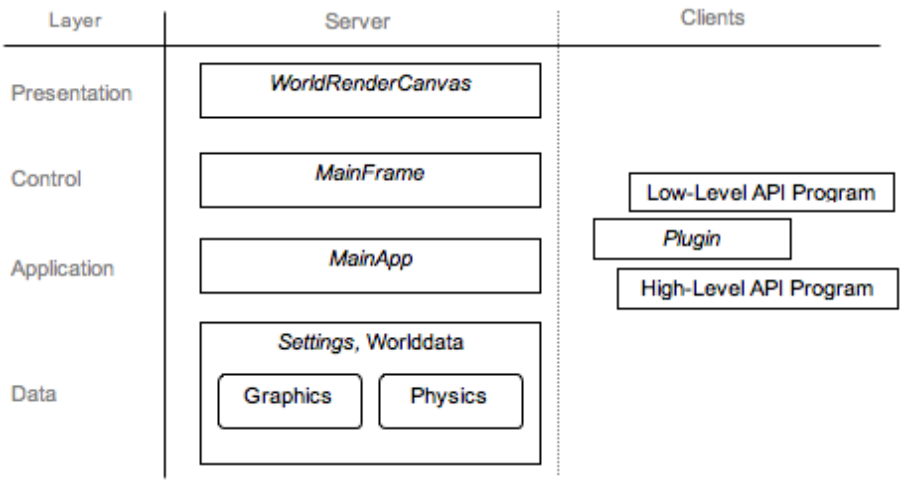

Figure 3.2: Overview of the layer architecture

The GUI consists of the presentation- and control-layer. Whereas the first layer keeps the functional parts for visualization, the second layer offers elements for controlling the program and simulation. This includes changing the users viewpoint, influencing the visualization or interacting with the simulation by changing execution speed or adjusting the error model parameters.

In the following, the thesis gives a closer look on the components of the system, especially on the server with the layer model, interfaces and implementation details. After that the focus is changed to plugins and controller programs followed by a discussion how those clients have to be structured and are used in SubSim. The EyebotPlugin is introduced as an example plugin implementation and an extension of the low-level API on the same.

4. Components of the Simulation System

4.1 The Server

A clear separation of the different parts of the simulation program, was the reason for a server layer model. The inner communication and data exchange happens through tight interfaces in between them. This results in a well structured and defined hierarchy between the program parts, represented as classes.

With the standardization of the layers, code dependencies are minimized and allow expanding or exchanging a program class or a whole layer easily. This abstraction is, especially for future development, a big advantage.

As the EyeBot controller programs are written in the programming language C, it was given to code the simulation program in C++. This has the advantage of being compatible to the controller program and implement an object-oriented design at the same time.

The SubSim program parts are several C++ classes based on classes of the application framework "wxWidget"[1]. This 3rd party library is an open source C++ framework with a easy-to-use API for writing GUI applications on multiple platforms. It offers not only GUI implementation with the typical windows widgets, multithreading but also some useful extensions like a proper *String* class or a error handling concept. SubSim especially takes advantage of the *wxGLCanvas*, an OpenGL canvas, for displaying the world and *wxDynamicLibrary* for handling clients, ie. a plugin or controller program.

To be highly customizable, SubSim uses configuration files for the application, simulation, world and worldobjects settings. These files use the XML syntax. XML stands for "eXtensible Markup Language"[van der Vlist 02]. It is a World Wide Web standard that allows information and services to be encoded with meaningful structure and semantics that computers and humans can understand. XML is great

[1] wxWidgets - http://www.wxwidgets.org

for information exchange and can easily be extended to include user-specified and industry-specified definition, so called ”tags”.

To get access to the settingfile data, SubSim processes these files with the help of the 3rd party library ”tinyXML”[2]. TinyXml parses an XML document and builds from that a Document Object Model (DOM) that can be easily read, modified and saved in the corresponding classes.

4.1.1 Layer Structure

The Application Layer

The application layer consists of one object of the class called *MainApp*. It is inherited from the wxWidget class *wxApplication* and represents the core of the program. This class provides the program initialization, plugin management as well the API implementation. As main layer, it connects the GUI with the data layer, too.

On program startup, *MainApp* initializes the application together with its settings and instantiates the GUI object. Additional startup parameters, e.g. the name of the simulation description file to load immediately, are parsed and processed. Then the user is able to start the simulation or load a simulation description file first if not happened yet.

The SubSim plugins are managed by a special plugin manager, which is part of the *MainApp* class. The manager loads and initializes the plugins once a simulation is loaded and keeps the plugins status and data. To load the plugin the class *wxDynamicLibrary* is used. Therefore the plugin manager is able to load a plugin, it has to be provided as a dynamic linked library (DLL), follow the SubSim plugin guidelines and specified in the application settings file. More details about plugins are given in chapter **??**.

With the classes *APILowLevel* and *APIPlugin*, SubSim offers two interfaces: One for clients to interact with the simulation via low-level function and one for plugins to access on high-level special functions such as camera processing, user in- and output, or querying the EyeBots HDT. As the *APIPlugin* includes the low-level API already, a plugin can access these functions without including the low-level API once more.

The API functions are the only connection between the external parts and the SubSim simulator. The only way for the program to influence the environment is through these functions and the only place for the simulator to influence the execution of the program is the implementation of these functions. This means, the simulation result is highly depended on the implementation of the API functions. Extensive testing is necessary therefore.

The usage of the two interfaces is different. As soon as a plugin gets loaded, it is associated with the class *APIPlugin* and has access to the SubSim functions since then.

Speaking of RoBIOS, all low- and high-level functions are defined static on the global namespace. This means the clients can immediately use the interface, without an explicit association. This has the advantage, that the controller program can be

[2]tinyXML - http://www.grinninglizard.com/tinyxml

used without changing the source code. But, as a unique connection is not available, the global static definition complicates the division between multiple running clients. So far, multiple active object are not supported, but it is considered in future development.

The Data Layer

The data layer holds all the data SubSim needs to execute and perform the simulation. SubSim requires certain information of the active object to be simulated and the environment in which the simulation should take place. The virtual world with its additionally passive objects and some other parameters like the visualization or physics engine settings have to be provided.

All configuration data is to be provided in several XML description files. There are four different types:

- Application description file
- Simulation description file
- World description file
- WorldObject description file

Figure 4.1 shows the overall structure and relations of the settings files. The dotted boxes are parts (XML tags) in the file. A solid line box is a reference to an external description file, whereas a folded corner box indicates an external XML file.

Through the XML syntax the files are quite intuitive and easy to create or edit. All the positional coordinates are specified in meters. Example XML files can be found in the appendix B of this thesis.

The application setting file has to be named `settings.xml` and stored in the same directory as the SubSim executable. It is independent from other settings files and is loaded on the startup of the simulator program. This file keeps the directory path to the world, object, GUI resource and plugin files. Graphic settings like screen width and height, as well a list of plugins to use in SubSim can be given.

A simulation file is the main configuration file for a simulation run and describes a scenario consisting of the environment and its objects. The file is needed in SubSim for starting a simulation and must have the suffix `.sub`.

First of all, the environment is described by giving the path to the world file. Additional active and passive objects are specified in a worldobject list, by giving the path to their description file. Active objects need a controller program and a HDT file in addition. Default settings for view, general physics settings and visualization can be described optionally.

A `*.sub` file can be loaded via the menubar of the GUI or given as startup parameter for loading immediately the scenario after initializing SubSim.

The world description file consists of the environment settings with its measures and graphic files. Depending on the specified heightmap and texture, this can be a pool

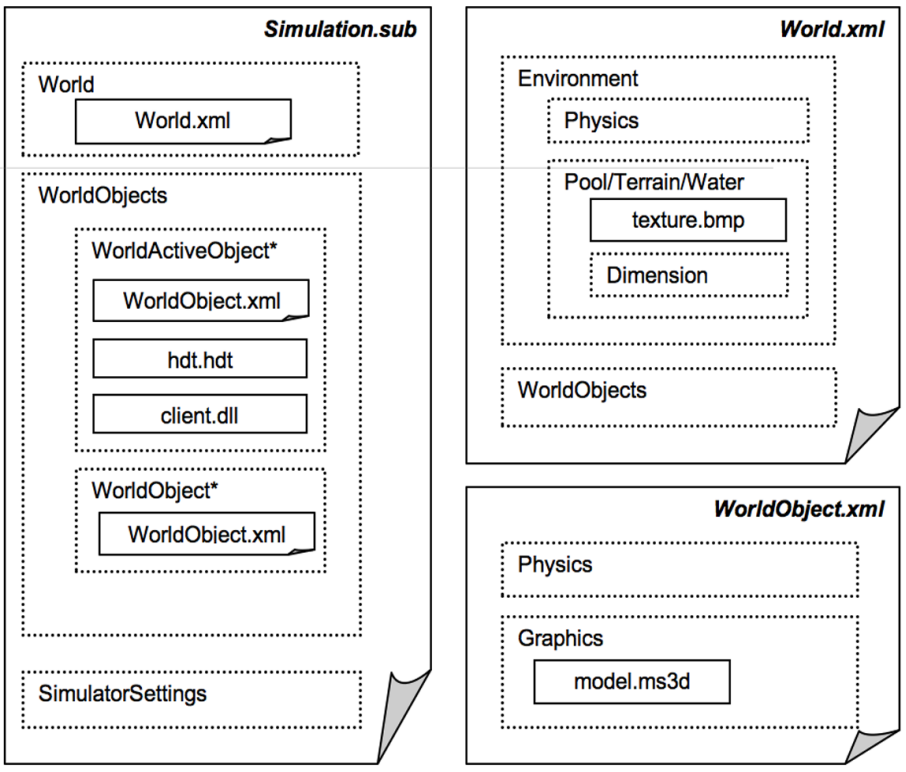

Figure 4.1: Overview of the description files

or an open terrain. A heightmap is a gray scaled bitmap file where white values indicate high points, and black values indicate low points (see figure 4.3).

The overall physical simulation parameters, such as gravity and water have to be defined. Default objects can be given like in the simulation file as a list of *WorldObjects*. The world description file can not be loaded directly from SubSim. Instead it has to be referenced in a simulation file once. There can be only one world file per simulation.

As a *WorldObject* consists of a graphical and a physical object, the XML file has to describe both parts. It controls the physical properties like mass, dimension and position, as well the connected actuators and sensors. The graphical description consists of the Milkshape 3D model file. In case this model differs from the physical object, scale and offset settings are to be given. A *WorldObject* turns into a *WorldActiveObject* by using the <*Submarine*> XML tag and specifying the path to its HDT file. Like a *World* file, a *WorldObject* file can not be directly loaded by SubSim and has to be referenced in a simulation file or world file instead.

Now, as the structure and content of the settings files is clear, the following part discusses their handling within SubSim.

One major part of the data layer is the class *Settings*. It purpose is to read and parse the application and simulation file for storing the data and supplying the program parts. The class follows the singleton design pattern [Rausch 04], as only one class instance is needed throughout the simulation program. The class is once instantiated and available on the global scope since then. This has the advantage that every class can access this class without the need of an explicit association.

With the data of the *Settings* class, the program core can build a *World* object. This happens by creating an instance of the class *World* giving the path to the worlds settings file. Additionally objects are added afterwards. The *World* class purpose is to hold all the information of the environment and perform the physical and graphical processing on its active and passive objects. It holds four different lists with all *WorldObjects*: First for objects in the background (like terrain), second for objects in the foreground (like water), third and fourth for passive and active objects. This differentiation is useful for dealing with the objects. As the fore- and background objects are not touched during simulation, all the active and passive objects can be accessed easily.

A *WorldObject* class itself consists of a *PhysicsObject* to link to the physics module and a *GraphicsObject* for linking to the graphics module. They are both compulsory and immediately created in the constructor of each object. The separation of the graphics and physics in different modules provides the ability to change and extend these parts easily in future development. This is a great advantage because especially the physics engine which is partly 3rd party software is still under development and may need to be exchanged or partially replaced in the future.

Depending on the kind of the object, different objects are inherited from the *WorldObject*. This can be a submarine as *WorldActiveObject* or a terrain and water as passive objects. Figure 4.2 gives an overview.

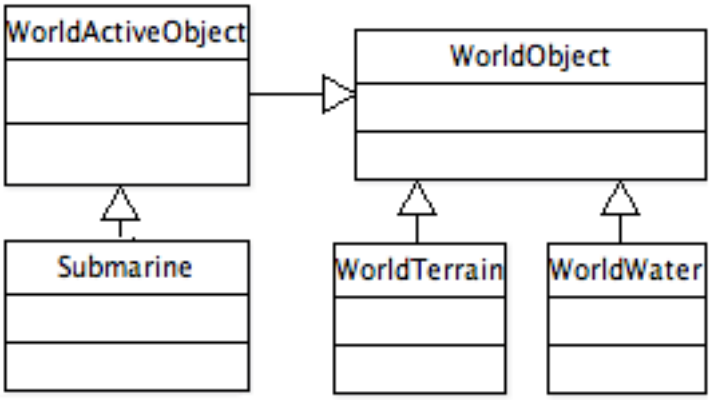

Figure 4.2: WorldObject class hierarchy

The environment terrain in SubSim is represented the *WorldTerrain* class as a heightmap. From these values terrain heights are constructed as graphical model, as illustrated in figure 4.3. Heightmap usage is commonly found in geospacial applications, allowing simulation of realistic terrain conditions by directly importing real world data.

A very special configuration is the *WorldActiveObject* class and in particular the *Submarine* class. For each simulated AUV one *WorldActiveObject* is created, extended for a *HDT* and extra hardware devices as sensors, actuators and cameras. A separate *ClientThread* runs and executes the EyeBot controller program. It does not matter which API the program is using. If the library the program is dynamically linked against is not available an exception is thrown. The separation of class

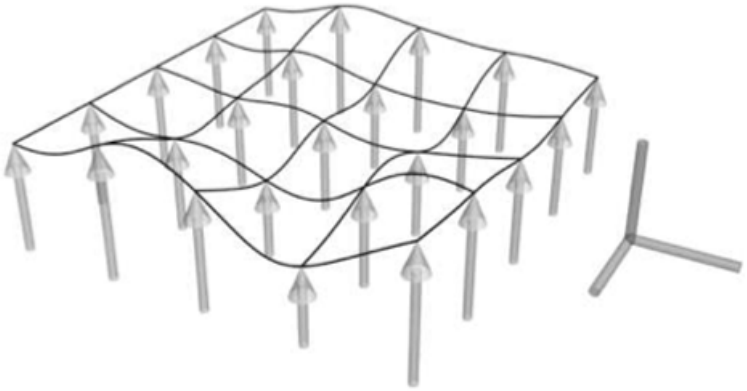

Figure 4.3: Heightmap Terrain

Submarine inherited from *WorldActiveObject* enables future introduction of other active objects than a submarine.

In the following the two major parts of an object, the physics and graphics module, are discussed:

The Physics Module

The physics simulation and calculation are outsourced in the 3rd party library "Physics Abstraction Layer" (PAL) with the physics engine "Newton". It is connected to the *WorldObject* via the wrapper class *PhysicsObject*.

Each *WorldObject* is physically simulated as a collection of bodies and geometries that represent the overall physical structure of the object. The physics module is responsible for calculating their position, movement and interaction with the surrounding environment. The effects of water, gravity, actuators, hydroplanes and collisions must be considered as forces applied on the body of the object.

One can say, the physics system is devided in two layers:

The first layer is the low-level physics. It consists of the basic physical simulation and is handled by the Newton package. It is responsible for keeping the bodies position, orientation and applying the forces to these bodies. The integration of Newton is straightforward according to the documentation and needs no special modifications. The simulation for collisions and water gravity is already implemented and can be used.

Second, the simulation of the various sensors and actuators is done in the high-level layer. This is the responsibility of PAL as well the communication to the Newton library in the low-level layer.

Sensors

Simulation of the sensors is a critical part of SubSim. It is essential to model sensor behavior in order to provide a realistic simulation environment for the AUV. SubSim offers a large range of positional, orientation and contact sensors. Most of the sensors can be directly modeled from the data available in the low-level physics. Every sensor is attached to a *WorldActiveObject* that represents a physical AUV model.

To present reasonable approximations to real sensors, the location of the sensors to the AUV must be defined by a position and orientation vector in the worldobjects settingsfile. The following list shows all sensors available in SubSim:

A Positional Sensitive Device (PSD) is a sensor to detect the distance from itself to the nearest object blocking its line of sight. Figure 4.4 illustrates the operation of a PSD sensor. A sensor ray is projected from its position (in this case the cube), in a specified direction (indicated by the arrow), until it reaches an object (the triangular prism). The distance between these two objects (ie: the length of the red line) is the returned value from the sensor.

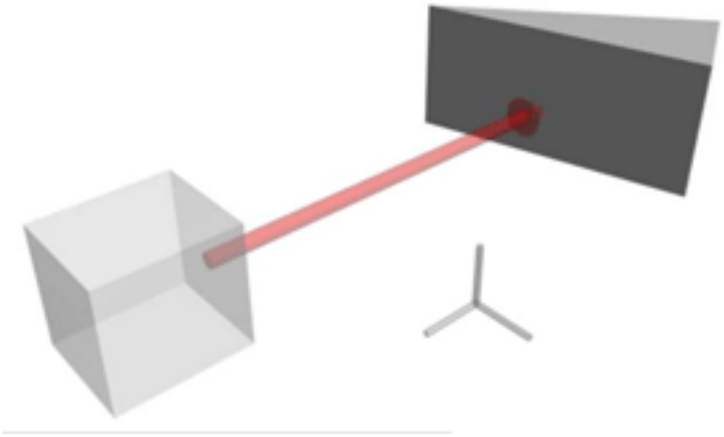

Figure 4.4: PSD Sensor

This sensor is simulated by using a ray-casting approach. A ray is cast from the PSDs position along the given direction until an intersection with another object in the environment is found. The sensor also has a maximum range at which it will indicate no intersection with any object. The low-level physics collision data structures are used to perform the ray casting. The maximum range and direction to cast to has to be described in the settings file.

An inclinometer sensor returns the angular orientation of a body. This sensor is simulated by extracting the orientation information of the attached body from the low-level physics. For that, the inclinometer sensor requires an input vector given in the settings file that, along with the position, indicates the plane about which the rotation angle is detected.

A gyroscope sensor is similar in its function to an inclinometer, in that it returns the angular velocity of a body. The gyroscope is implemented by either relying on extra velocity information available from the low-level physics library or by calculating the value from the body's previous angular positions. Like the inclinometer, the gyroscope requires an input vector for the plane to detect the rotation angle.

The compass sensor indicates the heading of a body relative to a global heading. This is simulated by an inclinometer restricted to the horizontal plane.

The velocitymeter sensor indicates the velocity at which the body is moving. The input vector given in the settings file indicates the direction in which the velocity is measured. This information can be extracted from the low-level physics layer. The

velocitymeter sensor will return the true velocity the body is moving at, and not the velocity relative to the water drag.

The contact sensor differs from the PSD sensor in that it detects any contact, and the location of the contact, that occurs on any given physical body. This allows the user to simulate simple bump sensors and also allows recording of locations where collisions occurred. This makes the contact sensor a powerful debugging and evaluation tool. The collision data structures of the underlying low-level library are queried when any contact occurs.

The camera, as the visual sensor, is not handled by the physics layer. Instead, the class *SubSimCamera* provides the data through rendering the environment and grabbing a frame. The procedure and handling is described in chapter 4.1.2

Actuators

The actuators provide the propulsion system of the submarine. They are responsible for active movement. SubSim provides two kind of actuators: one is a DC motor for moving links, such as a hydroplane, and another is a propeller for providing thrust to the underwater body.

The DC motor simply applies a rotational force to the body that it acts upon. There are two DC motor models exposed by SubSim:

A time-based model, typically used for controlling link torques:

$$T_A(t) = K_T \frac{V_\alpha(t) - K * \omega(t)}{R}$$

$T_A(t)$ is the torque
K_T is the motor torque constant (Nm/A)
$V_\alpha(t)$ is the applied armature voltage (V)
K is the motor back emf constant (Vs/rad)
$\omega(t)$ is the angular velocity of the motor (rad/s)

And a transfer function model, used for calculating the propellers angular velocity:

$$\frac{\theta}{V} = \frac{K}{(J+b)*(L+R)+K^2}$$

J is the moment of inertia of the rotor
b is the damping ratio of the mechanical system
L is the rotor electrical inductance
R is the terminal electrical resistance
K is the electro motive force constant

The first motor model is the standard armature controlled DC motor model, the second model is the armature motor model expressed as a transfer function. This allows a choice of model according to the users desired control output.

The propeller actuator is based on the lumped parameter dynamic thruster model developed by D.R. Yoerger et al [D. R. Yoerger 91].

The thrust produced is specified by:

$$Thrust = C_t * \Omega * |\Omega|$$

Ω is the propeller angular velocity
C_t is the proportionality constant

The Graphics Module

A graphical model of the *WorldObjects* is used to provide a user-friendly three-dimensional display. Additionally, it is needed to render and provide the image of a camera sensor.

The objects geometry is specified as a collection of boxes, spheres and cylinders. Each geometry object has its dimension, position and orientation. The combination of all geometries specifies the overall graphical body. All the geometries descriptions are imported through the Milkshape3D graphical model file. There are numbers of freely available editors and converters for this format allowing easy creation or conversion between various common CAD files.

The graphical model for MAKO AUV, developed in Milkshape3D, is shown in the next figures. Figure 4.5 shows the CAD view, as well the dimension measurements.

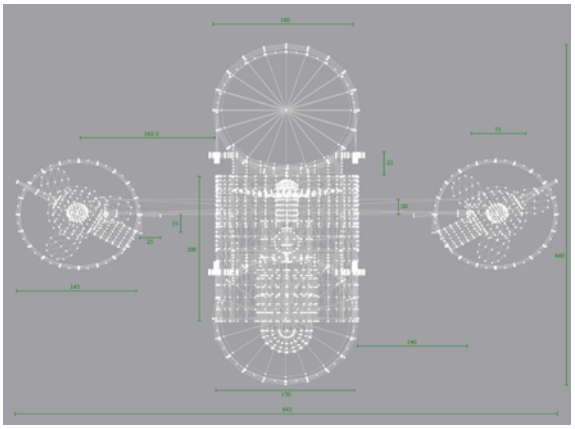

Figure 4.5: MAKO CAD model

Figure 4.6 shows the Milkshape3D rendered view of the model.

To display the graphical model in SubSim, the *WorldObject* is connected with the graphics module through the class *GraphicsObject*. The graphics module consists of a set of classes for handling the model drawings in OpenGL. The Milkshape 3D models with bitmap textures are loaded and rendered according to the users current viewpoint and the objects position and orientation.

Care should be taken that the graphical model reflects the physical model used for collision detection. As they are handled and described separately consistency of their measurements is not necessarily given. The scale attribute in the object description file is used to match the models.

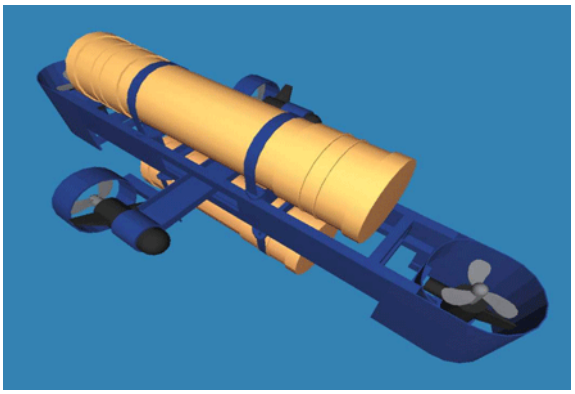

Figure 4.6: MAKO rendered model

The Presentation and Control Layer (GUI)

A Graphical User Interface (GUI) is a human-computer interface that uses windows, icons and menus, which can be manipulated by mouse and/or keyboard. Besides the visualization of the environment and giving simulation data, it enables the user to interact with the program and simulation. The GUI of SubSim is divided in two parts:

1. the presentation-layer which visualizes and displays data and
2. the control-layer which receives data input from the user.

SubSim uses classes and widgets from wxWidgets to build its GUI. The main window is an instance of the class *MainFrame* inherited from *wxFrame*. Additional windows can be included in a plugin or are *CameraFrames* to display images taken by cameras attached to a *WorldActiveObject*.

As seen in figure 4.7, the main window consists of the usual menubar on the top and statusbar along the bottom of the window frame. The menubar allows loading a simulation file, gives access to the plugin frames and the help system. The status bar is divided into several fields showing useful information of the simulation. There are fields for displaying generic text messages, number of running active objects, simulation time and render frame rate.

Besides that, the GUI consits of the World panel on the top left, the Control panel on the top right and the Info panel on the bottom.

The world panel is meant for displaying and controlling the simulation world. The main class is *WorldRenderCanvas* which shows the rendered environment with the objects according to the users viewpoint. It is inherited from the class *wxGLCanvas*, a canvas provided by wxWidgets to visualize OpenGL scenes. The user's viewpoint is controlled by a virtual camera, which takes continuous pictures of the scene. The cameras position, orientation and zoom are either controlled by mouse, keyboard or GUI sliders surrounding the canvas. A list of the command keys can be found in appendix E.

The control panel, located on the right, consists of widgets to influence the simulation, to change the cameras viewpoint or to switch the visualization of the simulation.

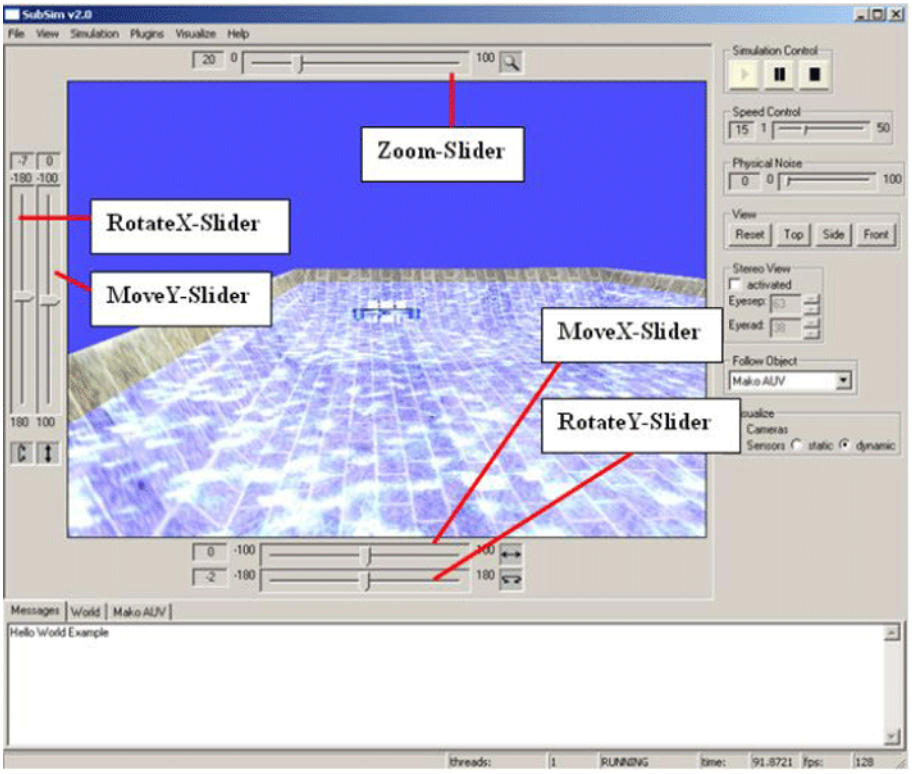

Figure 4.7: The SubSim main frame as Graphical User Interface (GUI)

The three buttons on the top, start, pause and stop the simulation. Whereas pause just interrupts the simulation, stop resets the simulation to its original beginning state. The simulation speed is adjusted with the slider beneath. It allows decelerating or accelerating the simulation time. The physical noise slider sets the error noise value of the physic engine.

Clicking the four view buttons changes the camera's viewpoint to a preset view. A choice can be made between front, top, side or initial view (settings from the simulation file).

The *WorldCamera* can be changed to a *StereoCamera* to see the *World* as a 3D anaglyph image. As the view is dependent on the users eye separation and angle, those parameters are adjusted with the *wxSpinControl* widget beneath.

By selecting an objects name from the dropdown box, the camera will start following this object. Sensors and cameras of all active objects can be visualized with selecting the checkboxes. The static option displays the sensors all the time, the dynamic visualizes the sensors only when the controller program reads from.

The Information panel is a *wxNotebook* widget and holds a set of pages to get information about the simulation, world or each active object.

One is the *Message Page*. It is the first page and consits of a messagebox. In here debug or information messages from the controller program are printed.

The second page is the *World Page* containing a table with all the object data such as name, current position and orientation. An object is manual placed by clicking in a cell and changing the value to the desired position or orientation.

The *ActiveObject Page* holds all the information from an active object. For every *WorldActiveObject* a special page is added to the information panel, displaying the HDT data and allowing access to the different cameras attached to the object. On the left side a table with the HDT data is placed. The table can be manually or automatically updated for monitoring the sensor values. If the object holds cameras, the user can look at the camera picture by clicking any of the view buttons for every camera on the bottom of the page. By doing so, this opens an extra frame of the class *CameraFrame* so the user has the opportunity to see the current image shot by the camera. This helps debugging if an additional view, like the EyebotPlugin LCD, does not display the current camera picture.

4.1.2 Rendering Procedure

To render a world, a camera is needed to set the viewpoint and perspective. After the viewpoint is set, the *World* class render function loops through all its objects calling their render functions. First the background objects like the terrain are processed. These are followed by the normal objects. This can be either a *WorldActiveObjects* or a passive *WorldObject* involved in the scene. Finally objects in the foreground are drawn. Once the render method of any *WorldObject* is called, its graphical object has to be updated first. This occurs through getting the current transformation matrix from the physics object. The matrix holds the position and rotation information which are applied to the graphical model. Afterwards the object gets drawn with the help of the graphics engine on the current OpenGL canvas.

The Camera Classes

As mentioned, a camera is necessary to set the scenes viewpoint or to get the image of a camera attached to a *WorldActiveObject*. A camera is represented by the class *WorldCamera* or inheritations of this class.

So far there are three different Cameras available:

1. WorldCamera
2. StereoCamera (inherited from WorldCamera)
3. SubSimCamera (inherited from WorldCamera)

The *WorldCamera* is the standard and main camera for visualizing the environment. It is meant as a usual camera and features changing its position, zoom and angles. In addition, a *WorldObject* can be associated which the camera follows.

The *StereoCamera* is meant for displaying a three-dimensional view. It extends the *WorldCamera* for an anaglyph picture. The image is produced by combining color-filtered images from two slightly different viewpoints into a single image. These images may then be viewed with ”anaglyph glasses”, which use color filters to moderate the light reaching each eye to create the illusion of a three dimensional image.

One of the most prominent features of SubSim is the capability to simulate a physical camera attached to a *WorldActiveObject*. This allows controller programs real-time image processing, e.g. for obstacle avoidance or target detection and tracking. The

camera implementation is realized with the class *SubSimCamera*, inherited from a *WorldCamera*. The camera position and angle are fixed and only changed by moving the object. To grab a picture and pass it to the controller program, a buffer, called p-buffer[PBuffer 04], for offscreen rendering is used. This buffer can be read using the EyeBot high-level API calls. The default camera functions have a resolution of 80*60 pixels (extended to higher resolutions) at 24bit color per pixel.

4.2 The Clients

There are two major types of clients: One is the plugin extending the functionality of SubSim and the other is the controller program controlling the active object. A controller program can again be divided in high- or low-level API usage.

The high-level API is EyeBot controller specific, and provides much more functions and easier handling of the hardware devices compared to the low-level API. On the other hand the low-level API is faster and more compatible to other types of controller besides EyeBot.

The following part details both types and introduces the *EyebotPlugin* as an example plugin and implementation of the EyeBot RoBIOS high-level API.

4.2.1 Plugin

A plugin is meant for extending the functionally and the GUI interface of SubSim. This may be an extension of its low-level API (as the *EyebotPlugin* does) or eg. adding special observers for data tracking. A plugin connects to the program core *MainApp* trough the interface called "APIPlugin", represented by the class with the same name. The plugin is automatically associated with this interface and its only connection for communication and data exchange with the simulator.

A plugin is a normal C++ library using the standard C functions as well all additional libraries SubSim offers. Its framework and GUI are, like SubSim, based on wxWidgets. To create a GUI, the class *PluginFrame* must be used.

In order SubSim can include and handle plugins properly, thier development has to follow some guidelines:

- The plugin must inherit from the class *Plugin* as well the GUI frame must inherit from *PluginFrame*.
- Make sure to provide a pluginname, version and description in the headerfile. This information is displayed to the user and helps track changes in plugins.
- Include the line DECLARE_DYNAMIC_CLASS(classname) in the plugin class declaration and the line IMPLEMENT_DYNAMIC_CLASS(class name, plugin) in the class implementation file. This is part of the wxWidgets dynamic class system and is needed so SubSim can create an instance of the plugin.

For including a plugin in SubSim it has to be compiled as a dynamic lilbrary (DLL) and specified in the plugin list of the application settings file. Several parameters can be given there also. On the next program start up, the plugin is automatically loaded and initialized. In case the plugin provides a frame, it can be viewed via the plugin menu in the menubar of the SubSim main frame.

EyebotPlugin

The EyebotPlugin is an example plugin for SubSim. It extends the simulator for the EyeBot RoBIOS high-level functions. This API provides special functions based on the SubSim low-level API. With that the user is possible to deal easily with the several sensors, actuators and cameras and perform data input/output through a GUI interface. The GUI interface is a frame with a 128 pixel width and 64 pixel height monochrome LCD and four inputkeys. It is provided through the class *EyeBotFrame* inherited from *PluginFrame*. All the text and graphic output is displayed like on the real EyeBot counterpart. To manipulate the display content, the high-level API offers three different ways:

- Printing characters or text
- Drawing simple shapes like lines and filled areas
- Fill the display with an image, e.g. an image grabbed from the camera

Two buffers handle the content and provide the output to the LCD: one textbuffer for keeping all the character input and one imagebuffer for drawing the shapes or grabbing the camera's image into.

To perform a keyboard input, the user has to focus the *EyeBotFrame* and click on one of the virtual keys. This invokes a key event stored in the keyboard buffer. The RoBIOS allows to read the keyboard buffer non-blocking or to wait for a specific key or any key to be pressed. Figure 4.8 shows the plugin frame with the LCD (1) and inupt keys (2).

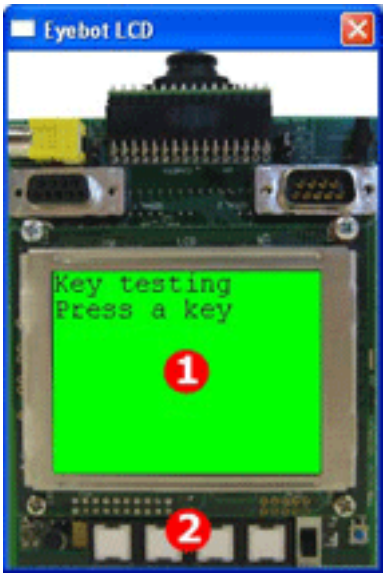

Figure 4.8: The EyebotPlugin Frame

Besides the EyeBotFrame, the actual API implementation is subdivided in several managers to keep a clear and logical separation of the RoBIOS function subsets. Figure 4.9 shows the class hierarchy with the manager classes of the plugin. The following list gives a short description of all managers:

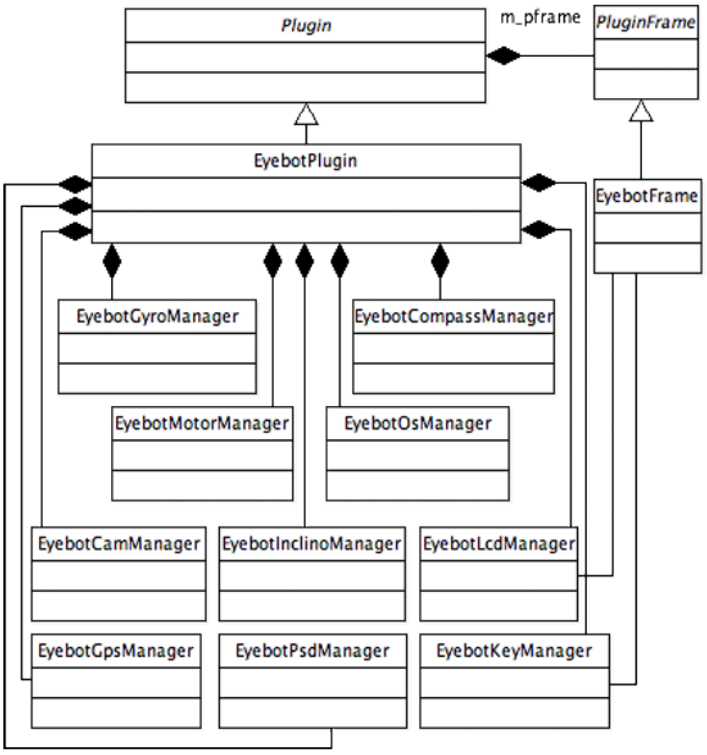

Figure 4.9: Class hierarchy of the EyebotPlugin

- The *EyebotCamManager* manages grayscale or a color camera functions for grabbing and manipulating images.
- The *EyebotKeyManager* manages the userinput via the four input keys in the EyeBotFrame.
- The *EyebotLcdManager* displays text and images on the LCD of the EyeBotFrame.
- The *EyebotMotorManager* manages initialization and communication to all actuators.
- The *EyebotOsManager* manages some basic functions for retrieving information like the ID of the active object, its name or the current time. For dealing with RoBIOS system, the timer-, multithreading- and data exchange functions are managed.
- The *EyebotPsdManager* manages initialization and communication to the PSD sensors.
- The *EyebotCompassManager* manages initialization and communication to the digital compass.
- The *EyebotGpsManager* manages initialization and communication to GPS sensor to get the current GPS position.

- The *EyebotGyroManager* manages initialization and communication to gyroscope sensor to measure the objects orientation.
- The *EyebotInclinoManager* manages initialization and communication to the inclinometer sensor for measuring inclination.

As the EyeBot plugin is an essential part of SubSim providing the RoBIOS high-level API, it is distributed with the simulator program and default loaded on program startup.

4.2.2 Controller Program

The controller program, to be simulated in SubSim, is the same program which is executed on the controller of a real AUV model. To execute this program in SubSim, no changes in the code are needed. The only difference to the usage on a AUV model is the program compilation and API linkage: It must be compiled as dynamic library (DLL) and linked against the SubSim library providing the low-level API or against a plugin containing the RoBIOS high-level functions headers.

The controller program file has to be defined with the HDT file in a worldobject settings file. SubSim treats the file as a client. Depending on the linkage it is connected to the simulation core trough the low-level API or the plugin providing the API linked against. In case of the RoBIOS high-level API this is the EyeBot plugin.

5. Example Simulation Case: Pipeline following

The following summarizes the project achievements with an example simulation case and demonstration of the program features. The simulation case targets a submarine MAKO controlled by an EyeBot controller moving along an underwater pipeline. It is an example for a common AUV application. Because of the deep and the underwater pressure it is too difficult and dangerous to send down scuba divers to the seabed. Instead AUVs are used to test pipelines or other objects for leaks, obsolescence and damage.

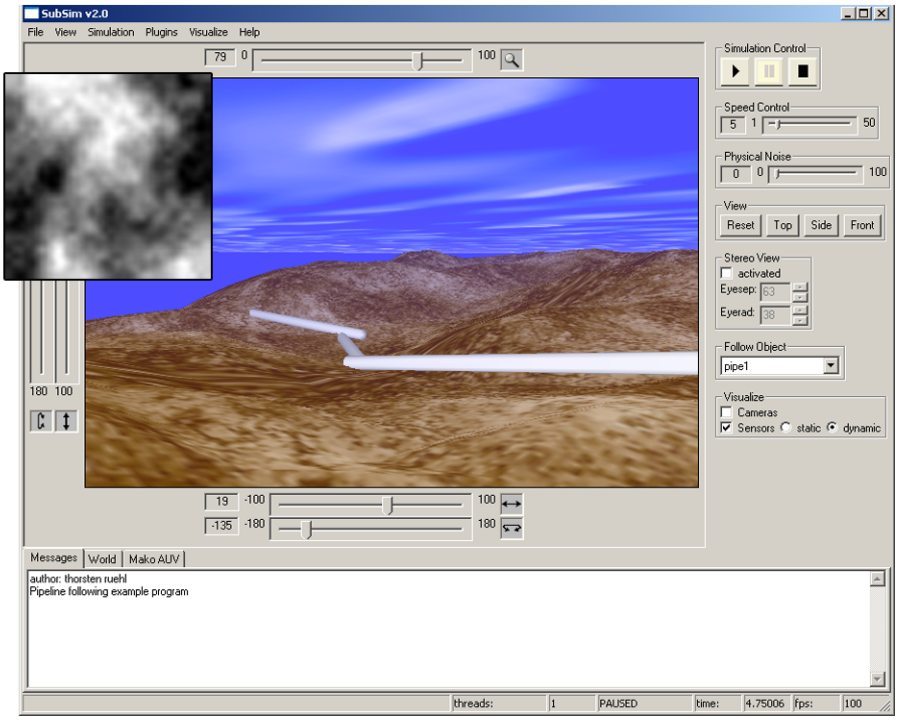

Figure 5.1: SubSim with the environment terrain loaded from a hightmap file.

In this example the down directed onboard camera takes continuously pictures of the seabed and pipeline. An image recognition algorithm on the controller processes the images. The submarine motion is adjusted to the determined pipeline position. Additional PSD sensors measure the distance to the object to keep a certain range.

The simulation environment is set up with the creation of the user defined terrain. The height information is taken from a heightmap bitmap file. Additional objects, in this case a pipeline of several sectors on the seabed, are placed afterwards. Figure 5.1 shows the terrain with the pipeline and the heightmap bitmap.

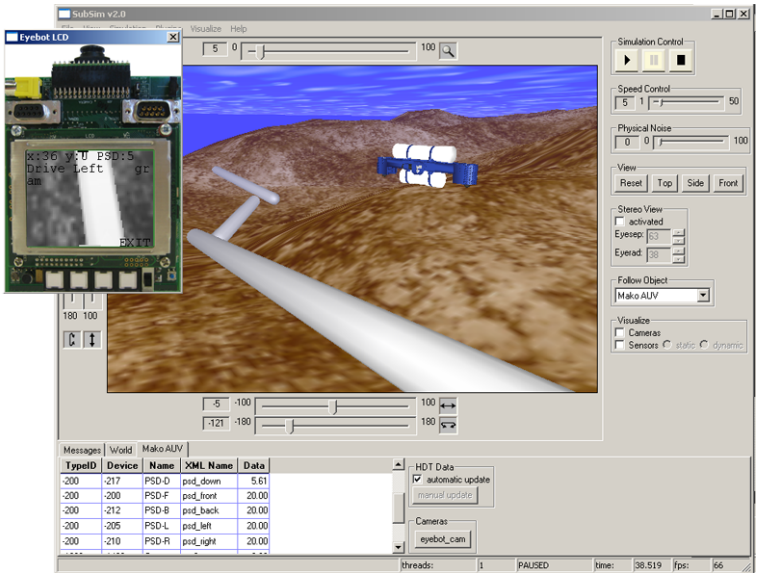

Figure 5.2: Submarine MAKO placed in the environment with the EyeBot Plugin frame loaded.

Once the environment is created, the active objects are added to the simulation. In this example the submarine model MAKO is put to the defined start position, as shown in figure 5.2.

The next step is to load the plugins required by the simulation. As the controller program uses the high-level API, the EyeBot Plugin is loaded and initialized. The plugin provides an extra frame to display the LCD output and to receive user input from the buttons.

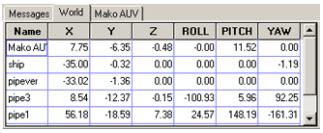

Figure 5.3: The World Page table with object coordinates.

As soon as all components are loaded, the simulation is ready to run. The user can start the simlation with the start button on the top right of the main window. The other buttons are to pause and stop the simulation. button. Whereas the pause button just interrupts the simulation, the stop button resets the simulation to its original beginning state.

The view of the simulation scene is dependent on the world camera's placement. Position, orientation and zoom are either controlled by mouse, keyboard or GUI sliders surrounding the world panel canvas. A list of the command keys can be found in appendix E.

During simulation time several parameters can be monitored for debug purposes. The information is combined on the bottom panel, subdivided in multiple pages for messages, world- and active object data.

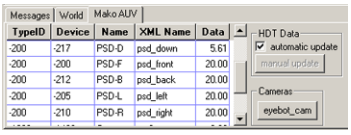

Figure 5.4: The ActiveObject Page with the HDT table and camera button.

Debug and information messages from the simulation or controller program are collected on the *Message Page*. The next page, called *World Page*, contains a table with all the object data such as name, current position coordinates and orientation as shown in figure 5.3. An object is manual placed by clicking in a cell and changing the value to the desired position or orientation.

The *ActiveObject Page* holds all the information from an active object. As shown in figure 5.4, the page displays the HDT data table and gives access to the different cameras attached to the object. The HDT table can be manually or automatically updated for monitoring the sensor values.

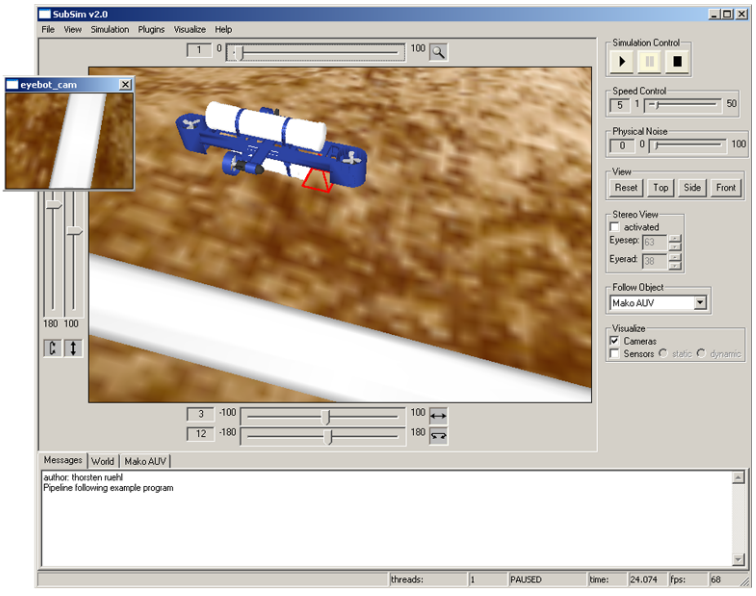

Figure 5.5: The camera frame showing the picture of a down directed camera represented as a red pyramid.

In case cameras are attached to the object, the user can look at the camera picture by clicking any of the view buttons for every camera on the bottom of the page. This opens an extra frame as shown in figure 5.5.

The simulation system supports visualization of PSD sensors and cameras attached to the object. The visualization can be switched on and off via the control panel or the menubar. Cameras of the submarine are presented as a red pyramid. According to the cameras orientation, the pyramid base points in the direction of the camera picture as shown in figure 5.5.

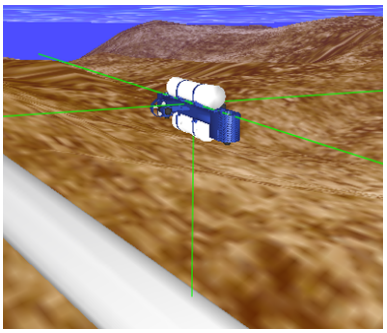

Figure 5.6: The submaine's PSD sensors visualized in static mode as a green line.

PSD sensors are visualized as a green line representing the sensors "line of sight". Two different modes are available: static or dynamic. Whereas the static option displays the sensors ray all the time, the dynamic visualizes the sensors only when the controller program reads from. Figure 5.6 shows the sensors visualized in static mode.

SubSim features as three dimensional view of the simulation scene. Is the "Stereo View" option activated, the visualization is changed to a 3D anaglyph image as shown in figure 5.7. The view is dependent on the users eye separation and angle. Those parameters are adjusted with the spincontrol widget in the control panel.

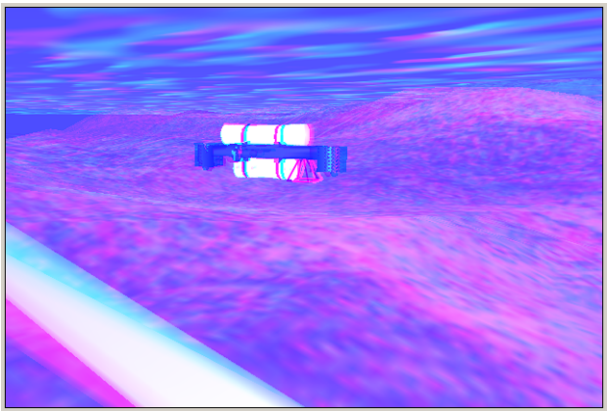

Figure 5.7: 3D anaglyph view of the simulation scene

6. Conclusion

This thesis project has achieved the following outcomes for the SubSim simulation system.

6.1 Achievements and Program Features

After four month of developing, program engineering and finally implementing the concept, the simulation program SubSim has reached the point of a first version to be released. The basic core functionality and its program framework are operating stable. The connection of the underlying physical and graphical models is finished. The structure of the settings files has been defined and is used in the simulation program.

To summarize, the simulator provides all advantages of a usual simulation environment:

- Run simulation in slow motion and time laps effects.
- Creating user-defined underwater environments in which the AUV acts in including different kind of objects to interfere with.
- Highly customizable through XML settings files for application, simulation, world and its objects .
- Plugin extension through interface for future expansions of SubSim functions and GUI.
- Low-Level interface for universal client programs.
- Implementation of the Eyebot RoBIOS high-level interface as EyebotPlugin.
- Fully support of a grey and color cameras.
- Basic physic engine suitable for underwater environments with the support of sensors and actuators simulation.

- Comprehensive documentation and helpsystem for SubSim enduser, plugin developer and for continuing work on the SubSim program.
- Various low- and high-level examples to demonstrate API usage

The SubSim Simulation program can be downloaded from:
http://robotics.ee.uwa.edu.au/auv/ftp.

6.2 Future Work

The simulator is finished to a usable and distributable version. But never the less, lots of work has to be done for a final version. Most effort has to be put in the plugin development. Due to the lack of time some of the EyebotPlugin high-level functions are merly insufficiently implemented. This may result in extending the plugin API to get access to more simulation data and functions.

SubSim will offer more functionality as more plugins are developed. Additional requirements were made. Data trackers for object positions and sensor data are requested. The tracked data, saved to a file, can be visualized and processed by an external program. Other plugins could extend SubSim for more hardware support like Bluetooth or WLAN.

On the long hand the simulation system can be opend for other robot models, different to submarines. Other types of robot models developed at the CIIPS groups are to be simulated.

A. Interface Description

APILowLevel

The low-level API consists of the following five functions:

```
SS_DID InitDevice( char* device_name );
SS_ERROR ReleaseDevice( SS_DID device );
SS_ERROR GetTime( SS_CLOCK *time );
SS_ERROR SetData( SS_DID device, DeviceData* data );
SS_ERROR GetData( SS_DID device, DeviceData* data );
```

The function `InitDevice()` initializes the device given by its name and stores it in the internal registry. It returns a unique handle *SS_DID* that can be used to further reference the device (e.g. sensors, motors). The function `ReleaseDevice()` releases the handle and device. Function `GetTime()` returns a time stamp holding the execution time of the submarines program in milliseconds. In case of failure an error code *SS_ERROR* is returned. The functions that are actually manipulating the sensors and actuators and therefore affect the interaction of the submarine with its environment are either the `GetData()` or `SetData()` function. While the first one retrieves the data (e.g. sensor readings), the later one changes the internal state of a device by passing control and/or information data to the device. Both return appropriate error codes if the operation fails.

APIPlugin

The Plugin API consists of the low-level API and the following functions:

`wxWindow* getParent();` - returns the pointer to *MainFrame*
`void CAMGetFrame(wxImage *image);` - grabs a frame from the current camera into *image*
`void* HDT_FindEntry(TypeID id, DeviceSemantics ds);` - finds a HDT entry
`int HDT_Validate();` - validates HDT

`int HDT_TypeCount(TypeID id);` - returns amount of a certain entry type
`char* HDT_GetString(TypeID id, DeviceSemantics ds);` - returns the name of a HDT entry
`DeviceSemantics HDT_FindSemantics(TypeID id, int k);` - returns semantics of a HDT entry

EyeBot high-level

For a complete list of the EyeBot RoBIOS functions please refer to
http://robotics.ee.uwa.edu.au/eyebot/doc/API/library.html

B. Description Files

Application File

```
<?xml version="1.0"?>
<Application name="SubSim">
   <Graphics>
      <Screen top="100" left="100" width="640" height="480" />
   </Graphics>

   <WorldPath>worlds/</WorldPath>
   <PluginPath>plugins/</PluginPath>
   <ResourcePath>resources/</ResourcePath>
   <ObjectPath>objects/</ObjectPath>
   <RobotPath>robots/</RobotPath>

   <Plugins>
      <Plugin file="eyebotPlugin.dll" name="EyebotPlugin" />
   </Plugins>
</Application>
```

An application settings file has to start and end with the `<Application>` tag. Different tags to set the graphical environment, default path and plugins are allowed.

The `<Graphics>` tag surrounds a `<Screen>` tag to set resolution and position of the GUI frame.

The list of plugins to load on program startup is given with the `<Plugins>` tag. The plugin itself is defined by the `<Plugin>` tag, with the attributes ”file” for the plugin filename and ”name” for the plugin name.

Simulation File - Example Wallfollowing

```
<?xml version="1.0"?>
<Simulation>
  <Docu text="Wallfollowing example"/>
  <World file="pool.xml" />

   <WorldObjects>
     <Submarine file="./mako.xml" hdtfile="./mako.hdt">
       <Origin x="3.0" y="0.0" z="3.0"/>
       <Client file="./wallfollowing.dll" />
     </Submarine>
  </WorldObjects>

  <SimulatorSettings>
       <Physics noise="0.002" speed="15"/>
       <View rotx="-20" roty="-20" strafex="-3" strafey="-11"
              zoom="10" followobject="Mako AUV"/>
       <Visualize psd="dynamic" camera="true"/>
  </SimulatorSettings>
</Simulation>
```

A simulation file has to start and end with the `<Simulation>` tag. In between different tags are allowed:

The `<World>` tags specifies the world file. This tag is compulsory and can be used only once.

The list of all world objects is defined with the `<WorldObjects>` tag. This tag must surround all world object entries.

Passive objects are defined by the `<WorldObject>` tag, active ones by the `<Submarine>` tag. The filename and path to the object's description file is given in the attribute "file". If no path is given, the default path for world objects files is used. To adjust position and orientation of the object, the `<origin>` and `<orientation>` tag is used as shown in the example.

Additional simulation settings are surrounded by the `<SimulatorSettings>` tag. There are settings for physics engine (`<Physics>`), viewpoint (`<View>`) and visualization (`<Visualize>`).

World File - Example Pool

```
1 <?xml version="1.0"?>
2 <World>
3    <Environment>
4         <Physics>
5           <Engine engine="Newton" />
6           <Gravity x="0" y="-9.81" z="0" />
7         </Physics>
8         <Water>
9            <Dimensions width="24" length="49" />
10           <Texture file="water.bmp" />
11        </Water>
12
13        <Terrain>
14          <Origin x="0" y="-3" z="0" />
15          <Dimensions width="25" length="50" depth="4" />
16          <Heightmap file="pool.bmp" />
17          <Texture file="stone.bmp" />
18        </Terrain>
19   </Environment>
20
21   <WorldObjects>
22        <WorldObject file="buoy/buoy.xml" />
23   </WorldObjects>
24 </World>
```

A world file has to start and end with the `<World>` tag. It is broken up into two aspects: The environment (`<Environment>`), and objects (`<WorldObjects>`).

The world environment consists of the overall physical simulation parameters, a water and terrain description..

First, the physics engine and parameters, like gravity, are to defined. Second, the origin, dimension and texture of the water and terrain are specified in the `<Water>` and `<Terrain>` tag.

The list of world objects follows the same description as in the simulation file.

WorldObject File - Example Buoy

```
<?xml version = "1.0"?>
<WorldObject name = "buoy">
   <Origin x="8" y="5" z="0" />

   <Graphics>
      <Origin x="0" y="0" z="0.0" />
      <Scale x="0.05" y="0.05" z="0.05" />
      <Model file="buoy.ms3d" />
   </Graphics>

   <Physics>
      <Primitives>
        <Sphere name="buoy">
           <Position x="0" y="0" z="0"></Position>
           <Dimensions radius="1.0"></Dimensions>
           <Mass mass= "0.9"> </Mass>
        </Sphere>
      </Primitives>

      <Actuators>
        <FakeBuoyancy name="buoy_water">
           <Connection connects="buoy"></Connection>
           <Density density="0.99829"></Density>
        </FakeBuoyancy>
      </Actuators>
   </Physics>
</WorldObject>
```

A passive object file has to start and end with the `<WorldObject>` tag. It consists of two parts: one is the graphics part, indicated by the `<Graphics>` tag, the other is the physics part, specified by the `<Physics>` tag.

For the graphics, the `<Origin>` tag is supported to define the object's position. The graphic model file is specified with the `<Model>` tag and fitted to the physics model by the `<Scale>` tag.

The `<Physics>` tag contains the model's physical representation. The primitive spheres with their position, dimension and mass are defined.

WorldActiveObject File - Example AUV ”MAKO”

```
<?xml version="1.0"?>
<Submarine name="Mako␣AUV">
  <Origin x="10" y="0" z="0"/>

  <Graphics>
        <Origin x="0" y="0" z="0"/>
        <Scale x="0.18" y="0.18" z="0.18" />
        <Model  file="mako.ms3d" />
  </Graphics>

  <Physics>
    <Primitives>
      <Box name="Mako␣AUV">
        <Position x="0" y="0" z="0" />
        <Dimensions width="1.8" height="0.5" depth="0.5" />
        <Mass mass= "0.5"> </Mass>
      </Box>
    </Primitives>
    <Sensors>
         ...
    </Sensors>
    <Actuators>
         ...
    </Actuators>
  </Physics>
</Submarine>
```

An active object file has to start and end with the `<Submarine>` tag. The tags are the passive object tags with an extended physics part. In here, sensors and actuators are additionally defined by tags according to their name. For example, a PSD sensor is specified by the `<PSD>` tag.

C. SubSim Class Hierarchy

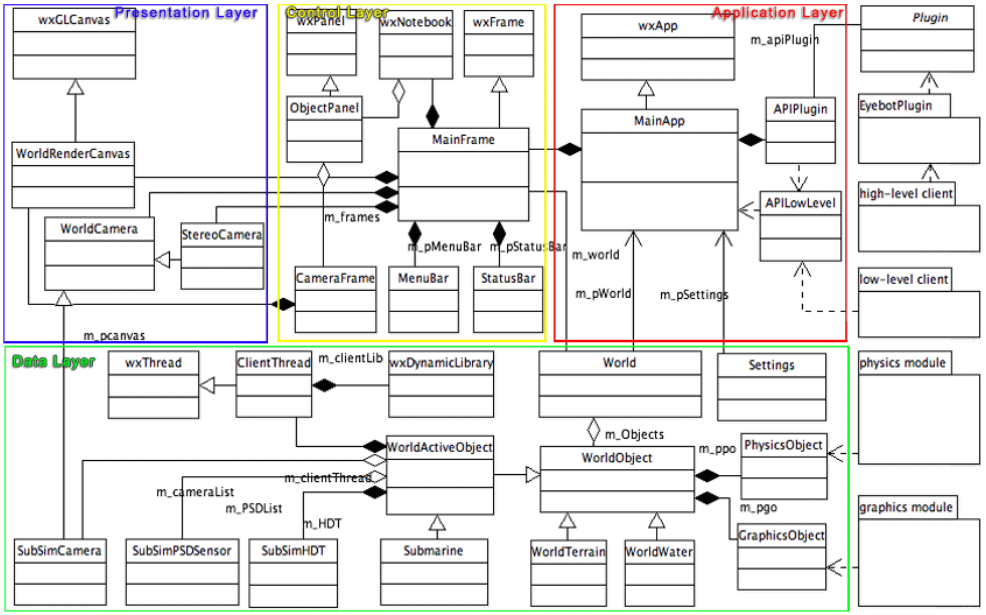

Figure C.1: The SubSim class hierarchy

D. SubSim Manual

The following brief introduction to the usage of the simulation software helps to understand how to write and use a controller program in SubSim. In order to simulate a controller program in SubSim, the program has to be created and compiled first. Next step is to set the simulation environment. When everything is ready, SubSim is able to simulate the controller program.

Creating the Controller Program

The easiest way to create a controller program is to copy the whole folder named "HelloWorld" in examples directory. This has the advantage that all necessary file are already created and only some minor adjustments are needed. Depending on API to use, choose the high- or low-level API example.

The folder consists of a `makefile`, a `helloworld.cpp` and a `hellowor- ld.sub` file. As it is a generic makefile, no further changes are needed for this file. The other files can be renamed as wished. It is recommended to use same name for folder, `*.cpp` and `*.sub` file for indicating their relation. The `*.cpp` controller program can be written with any text editor or development IDE. The generic texteditor "jEdit"[1], together with its "ConsolePlugin", has the capability to compile the C++ controller program easily within the editor. This is quite handy a is recommended to develop own programs.

Assure the right API file is included: Use `APILowLevel.h` when developing a low-level program, `API-eyebot_highlevel.h` for a high-level program. Mind that the high-level API has much more functionality but needs the EyebotPlugin loaded and is slower than the low-level API. A list of the function for both API can found in the appendix A of this thesis.

Compiling the Controller Program

As compiler, the gcc compiler from the package MinGW[2] is recommended. MinGW is a collection of freely available and distributable Windows specific header files

[1] jEdit - http://www.jedit.org
[2] MinGW - http://www.mingw.org

and import libraries combined with GNU toolsets that allow one to produce native Windows programs that do not rely on any 3rd party C runtime DLLs.

To compile your program, change the console prompt to the working directory and type `make`. This executes the makefile located in the directory, compiles the controller program and creates a `.dll` file useable with SubSim.

Setting Up

Next step is setting up the simulation environment. This happens by defining the several settings files. The major file is the `*.sub` file located in the directory. The world with its objects and some default settings have to be specified. Add a submarine to the object list and give the path to the controller program as DLL file in the client tag. Several example world and object files are provided with SubSim and can be used. To create new ones, take these as sample files and adjust them to your needs.

Starting the Controller Program

To finally simulate the controller program, start SubSim and load the `*.sub` file from the filemenu in the menubar. Alternatively, a double click on the `*.sub` file within the windows explorer invokes SubSim to start with the subfile immediately loaded. Once the simulation is loaded clicking the start button on top right starts the simulation.

A more detail description can be found in the enduser documentation provided with SubSim. This can be accessed via the helpmenu in the menubar. Sample setting files for the application, simulation, world and object can be found in the appendix B. These samples provide a description for all possible parameters and default values.

E. GUI Keycommands

Zoom Slider: Get closer to the object with the camera	SHIFT+CTRL+UP SHIFT+CTRL+DOWN
Strafe Y: Move the camera in Y-direction (up and down)	UP DOWN
Rotate Y: Rotate the scene around the Y-axis	CTRL+UP CTRL+DOWN
Strafe X: Move the camera in X-direction (left and right)	LEFT RIGHT
Rotate X: Rotate the scene around the X-axis	CTRL+LEFT CTRL+RIGHT

Bibliography

[Bräunl 01] Thomas Bräunl. *Scaling Down Mobile Robots - A Joint Project in Intelligent Mini-Robot Research*. IEEE J. Oceanic Eng., 2001.

[D. R. Yoerger 91] J. E. Slotine D. R. Yoerger, J. G. Cooke. *The Influence of Thruster Dynamics on Underwater Vehicle Behavious and Their Incorporation Into Control System Design*. IEEE J. Oceanic Eng., 1991.

[Dave Shreiner 05] Jackie Neider Dave Shreiner, Mason Woo. *OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley Professional, 2005.

[PAL 05] pal - Physics Abstraction Layer, 2005. http://pal.sourceforge.net.

[PBuffer 04] PBuffer Rendering, 2004. http://www.mensa3d.org/pbuffers.html.

[Rausch 04] J. Prof. Dr. Rausch. *Grundlagen des Software Engineering*, Kap. 10. Technische Universität Kaiserslautern, 2004.

[Ridao 04] P. Ridao, E. Batlle, D. Ribas, M. Carreras. NEPTUNE: A HIL Simulator for Multiple UUVs. In *Oceans04 MTS/IEEE*, Kobe, Japan, November 9-12 2004.

[SIM 05] Wikipedia, the free encyclopedia - Simulation, 2005. http://en.wikipedia.org/wiki/Simulation.

[Thomas Bräunl 00] B. Graf Thomas Bräunl. *Small Robot Agents with On-Board Vision and Local Intelligence*. Advanced Robotics, 2000.

[van der Vlist 02] Eric van der Vlist. *XML Schema*. O'Reilly Media, 2002.