WAVESIM - WATER VEHICLE SIMULATOR

António Santos **Aníbal Matos**

{*ansantos, anibal*}*@fe.up.pt*
Ocean Systems Group
Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias
4200-465 Porto, Portugal

Abstract: This article introduces a new framework for the simulation of multiple water vehicles operating simultaneously. Its main goal is to be used as a testing tool to help the development and implementation of the vehicles' on-board software. This simulator provides real-time realistic dynamic behaviour of underwater and surface vehicles as well as acoustic navigation by simulating acoustic signal propagation and providing acoustic transducers on vehicles and pre-located buoys. Pressure, temperature, condutivity, salinity and sonar sensors emulation is also provided and water flows can also be defined. There should be no intrusion in the existing software and a small SDK is available to help building a broker application that makes the bridge between the software and the simulator, both possibly running on different network peers. A provided 3D visualization system may also be used to control and watch the simulation.

Keywords: Simulation, multiple vehicles, acoustic navigation, dynamics

1. INTRODUCTION

No matter what we're talking about, testing and debugging is the most time consuming task in the development process. In what it takes to ocean technology stuff, this statement is even more stressed due to sea related difficulties that arise when setting up the needed apparatus to get your device properly working and operating on water. Surface and underwater vehicles construction is usually very complex. Not only the physical aspects of the vehicle must be very tight and well thought-out but also the electronic components and the software must be designed in order to safely and successfully accomplish several kinds of missions. Testing the software usually means taking the vehicle into water which sometimes takes a lot of time and effort. So, the possibility of testing the on-board control and navigation software in the lab with minimal intrusion is welcomed by the community. Of course, the simulator should provide a realistic environment so that the software can behave as close as possible as it would in a real scenario. The dynamic properties of the vehicle, the acoustic navigation and the on-board sensors are the prevailing factors the on-board software is depending on. There are few similar applications in this area. The *NPS AUV Workbench* (NPS, 2006) and *SubSim* (Braünl, 2006) are probably the most well-known simulators but their main goal is different from *WaVeSim*'s. Mainly, the possibility of testing the on-board software, the support for multiple vehicles connected by network and operating simultaneously and the support for real-time acoustic navigation are the features that greatly distinguish *WaVeSim* from the others and the major motivation factors to start this project.

2. *WAVESIM*

The Ocean Systems Group at the Faculty of Engineering in the University of Porto is developing *WaVeSim*, a water vehicle simulator written in C++ which takes into account all those testing and debugging issues and will hopefully ease the pain of setting up a camp site for software testing operations. Those who have properly developed control and navigation software that communicates with the hardware through its drivers can easily use this simulator by just developing a broker which will replace the drivers and will do all the interaction with *WaVeSim* through an ubiquitous medium like the Internet and by using a clear and specific communication protocol. The main idea is to run the on-board software in the vehicle but with the sensors turned off. That is, the on-board software will not be aware that it is working with a simulator instead of a real set. This broker can be easily developed by using a provided library.

3. MAIN FEATURES

This kind of simulator is not that common in this area. *WaVeSim* has some major features that should appeal to the people working with this technology. First, it supports simultaneous multiple vehicle simulation. Several surface or underwater vehicles can be connected and operating simultaneously and the simulator should provide realistic behaviour for each vehicle independently. Second, it provides realistic dynamics and hydrodynamics by using a physics engine. The properties of each vehicle must be provided to the simulator, like its geometry, mass and inertia, as well as hydrodynamic coefficients so that everything can be correctly integrated and the vehicle movement may be as realistic as possible. Third, it precisely emulates some common sensors like temperature, pressure, condutivity and sonar sensors. This sensor emulation is done by external processes so that it may be easy and practical to hot-swap them. Finally, this simulator is also able to simulate acoustic navigation by allowing the positioning of virtual acoustic beacons in the simulation world and by coupling emulated acoustic devices to the simulated vehicles and buoys. This way, the developed algorithms for acoustic navigation can easily be tested without setting up a real acoustic network.

4. GENERAL OVERVIEW

Like most of the newest projects that are being actively developed, *WaVeSim* relies on the Internet. The communication between the on-board software and the simulator is based on a message protocol which uses UDP over the network for transportation. *WaVeSim* accepts client connections anytime and provides a keep-alive system to check if some client has disconnected. All the simulation data is kept in semaphore protected shared memory areas so that it can be available to external processes. Among these are the sensor emulators which update their values based on the vehicle's position. Also, the visualisation system is decoupled from the simulator itself so that the latter may run using a different visualisation system or with none at all. An XML file-based configuration system allows several parameters to be easily defined. The broker application will replace the real hardware drivers and will take over their communication channels with the main control and navigation software. Then, it will do all the necessary interaction with *WaVeSim* and will translate the vehicle's data that comes from the simulator into properly formatted data that can be sent to the main software like a real device driver would do.

5. INTERNAL STRUCTURE

WaVeSim's internal structure is based on threads, each one with a specific task. There are a few global data structures which are shared by the simultaneous running threads, mainly vehicles, buoys and acoustic signals maps and vectors. Two shared memory areas are used to store simulation data. One of them stores general simulation data like the elapsed time, the simulation state, the number of clients and some client data like each client's type, connection status and keep-alive information. Also, the configurable parameters are also stored here like the timestep, the sound speed, fluid density, water flows and other simulation world related data. The other shared memory area stores vehicles and buoys' data. Besides the dynamic data like geometry, position, attitude, angular and linear velocities and motor actuation, also the output from the several sensor emulators is stored here. Also, a few other client specific data is stored here like its ID, name, model, IP address and port. These areas are protected by POSIX semaphores to guarantee data consistency when different processes or threads are performing I/O operations on them.

5.1 Simulation timer

WaVeSim's main process has several components that interact with each other so that a real-time simulation can be achieved. In the deep core of the system lies a timer which fires an event at each pre-defined timestep. The precision of this

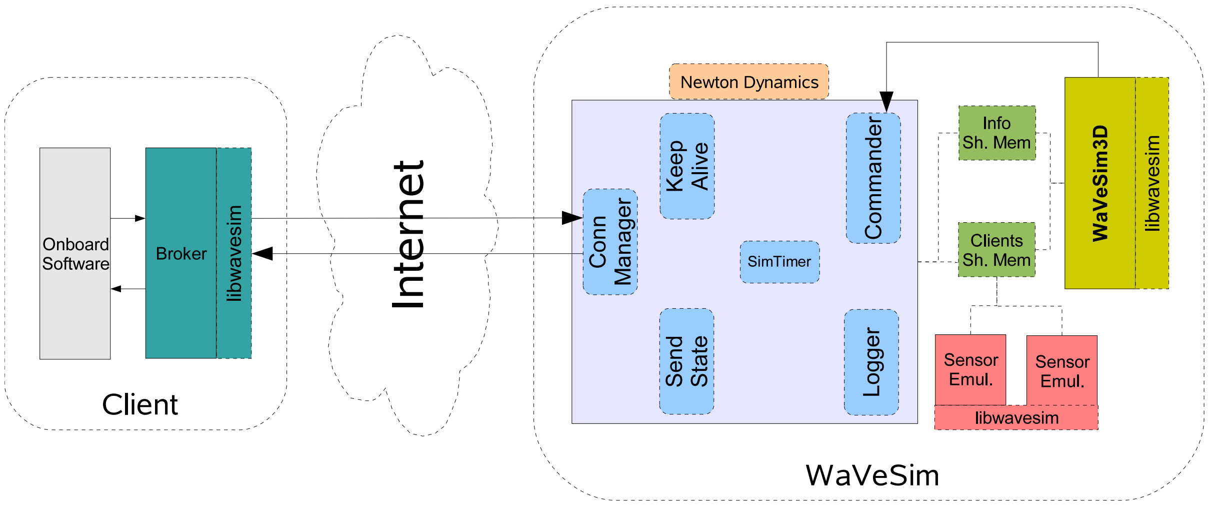


Fig. 1. *WaVeSim* general overview

timestep is limited by the timer frequency value set in the kernel, but most Linux distributions allow a precision of 4 ms or less. At each timestep advance, all the vehicles' dynamic data and the active acoustic signals' properties are updated. When the simulation is stopped, this timer is unarmed and there no updates at all, eventhough the clients still may interact with the simulator. The simulation timestamp is simply the number of tenths of milisecond since the start of simulation. This value is computed by simply calculating the difference between the actual system clock time and the one from back when the simulation had started.

5.2 *Connection Manager*

The connection manager runs as a thread. It opens a single UDP socket and listens for incoming messages on it. These messages' format obey to a predefined and very clear messaging protocol and this manager reacts accordingly to each message type. Also, this manager is used to send any message to a client by using its IP address and port.

5.3 *Commander*

The commander is simply a POSIX message queue that is used to receive command messages from the outside. There's a short set of command messages available, mainly for starting/stopping the simulation.

5.4 *SendState and KeepAlive*

These are two threads that perform two of the most important tasks in *WaVeSim*. *SendState* periodically cycles through all the clients' data structures and sends them to each client individually. Since *WaVeSim* uses stateless connections, *KeepAlive* keeps track of the connection status of each client to be sure that when a client closes the socket on its side it is removed from the simulation. This keep-alive system works by periodically sending a simple message to each client and measuring the time it takes to receive the reply. If it takes more than a pre-defined number of seconds, then the client is considered disconnected and is removed from the simulation.

5.5 *Logger*

A simulation shouldn't be limited to its running time. In order to get the most of it, it must be possible to analyse all the elapsed events at a later time, mainly the vehicles' movements and the exchanged acoustic signals. Therefore, a logging system is also running as a thread, logging the vehicle's position, attitude, linear and angular velocities and motor actuation at a rate defined by the user. Apart from that, all the sent and received signals are logged with their frequency and intensity at the time the event happens.

6. DYNAMICS

The on-board control and navigation software acts upon the vehicle's position and attitude in order to take it to a specific waypoint. The motor actuation forces, buoyancy and drag forces determine the vehicle's movement, so for the simulation to be really useful we need to have a fast and precise dynamics calculation. *WaVeSim* uses the *Newton Game Dynamics* (NGD) (Jerez, 2006) physics engine to take care of all this stuff. This physics engine is an integrated solution for real-time simulation of physics environments. It implements a deterministic solver for the dynamic equations based on a timestep advance. It features a quite extensive API so that it can easily be integrated with any C/C++ application. NGD works by creating a world defined by its boundaries and by adding bodies to it at initial positions. Each body has its own geometry based on geometric primitives and has a mass matrix and a transformation matrix. In order to apply forces to it, a callback function should be provided so that NGD calls it for each timestep advance. Finally, we need to provide several vehicle dynamic properties to ensure that the physics engine gives the results we expect. In order to have things quite simple, *WaVeSim* uses basic geometric primitives for each client type: surface vehicles are boxes, underwater vehicles are cilinders and buoys are spheres. For each client type, *WaVeSim* allows the definition of several different models, each one with its own characteristics. For each model we should define the following dynamic properties that should be as close as possible to the ones from the vehicle we want to simulate (Fossen, 1991):

- Depth, width and height (for surface vehicles)
- Radius and length (for underwater vehicles)
- Diameter (for buoys)
- Mass and added mass
- Inertia and added inertia on each axis
- Center of mass
- Linear drag coefficients (first and second order)
- Angular drag coefficients (first and second order)
- Thrusters position and maximum force (not available on buoys)

For each body, the callback function that applies forces and torques acts like the following (Fossen, 1991):

(1) Get vehicle's velocity from NGD and subtract to it the water's velocity at that point.
(2) Apply buoyancy force (NGD supports it by defining a water plane and applying the Archimedes principle).
(3) Apply gravity force.
(4) Apply linear drag ($F_d = \sum_n a_n v|v|^{n-1}$).
(5) Apply angular drag ($\tau_d = \sum_n a_n \omega|\omega|^{n-1}$).
(6) Apply the sum of all motor forces on the local xx axis (horizontal motors) and on the local zz axis (vertical motors (underwater vehicles)).
(7) Apply the sum of all motor torques based on each motor's location and the vehicle's center of mass.

NGD also handles body collisions. The vehicle's transformation matrix (with orientation and position) is then stored in the vehicles' shared memory area.

7. ACOUSTIC NAVIGATION

This simulator is very specific to robotic devices operating in water environments. Thus, since acoustic navigation is the preferred navigation method, *WaVeSim* supports signal propagation and acoustic transducer emulation on vehicles and buoys. If a vehicle has to have an acoustic transducer, that must be declared in its configuration. If so, it also must be declared its position on the vehicle. That made, the vehicle can send and receive acoustic signals. When the on-board software fires an acoustic signal request, the simulator gets the message and adds the signal to a vector of active signals. Each signal has a frequency, initial and actual intensity, radius, timestamp and origin point. For each timestep advance, each active signal's intensity and the radius of the signal's propagation sphere are updated. The radius is calculated with

$$r = c(t - t_0)$$

where c is the sound speed in water, t_0 the signal's emission timestamp and t the actual timestamp. The actual intensity is calculated with (Coates, 2001)

$$I = I_0 - (20\log_{10}(r)) + (0.005r)$$

The signal is considered dead when its intensity is below a pre-defined ambient noise level. Also, by computing the distance between the signal's origin point and each vehicle's acoustic transducer (if any), the simulator knows when that vehicle should be notified of an incoming acoustic signal. To have an even more realistic behaviour of the virtual acoustic transducers, the user should also set a signal reception intensity threshold (only accepts signals with intensity above this value), a delay time (the time the circuitry takes to reply to a query signal) and a recharging time (the idle time after sending a signal).

8. SENSOR EMULATION

Most control and navigation algorithm decisions are based on the output from several sensors. While positioning and orientation data is directly given by NGD, other kind of sensor data must be generated to fulfill the need of those algorithms. Usually, underwater vehicles are armed with, pressure, condutivity, temperature and depth sensors, most of the times grouped in a CTD sensor. So, *WaVeSim* allows external sensor emulators to be running and updating their values in the clients' shared memory area. As for now, *WaVeSim* supplies emulators for all those sensors mentioned above. They compute their output by checking the vehicle's position and calling a function to produce the final value.

8.1 Pressure

The pressure sensor emulator calculates the pressure value based on the vehicle's depth (z):

$$P(z) = P_0 + P_1 z + P_2 z^2$$

P_0 is the usual atmospheric pressure while P_1 and P_2 are provided coefficients that define the pressure profile.

8.2 Temperature

The temperature sensor emulator tries to capture the usual underwater temperature profile. So, to simplify the definition of parameters, temperature is measured by segments, using the behaviour in Fig. 2. Therefore, we need to provide the

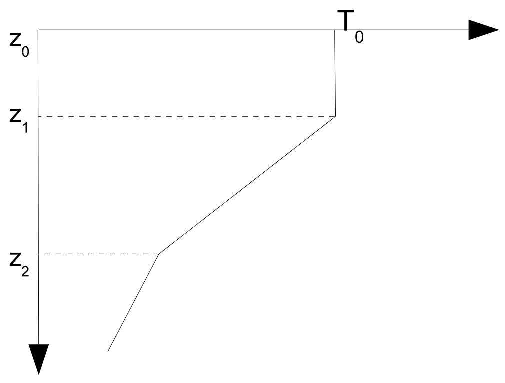


Fig. 2. Underwater temperature standard profile.

temperature at water surface (T_0), depths z_1 and z_2 and the derivative values of the linear function between z_1 and z_2 and between z_2 and the bottom. However, there's little interest in always following this same behaviour at every coordinate. We need to provide a way to map some temperature spots which are due to external factors like, for example, pollution. So, to the temperature value obtained before, we should add the deviation made by the spot centered at (x_i, y_i, z_i). Finally we have:

$$\begin{aligned} T(x,y,z) = &T(z) + \\ &+ \sum_i \Delta_i e^{-[\alpha_i^2(x-x_i)^2+\beta_i^2(y-y_i)^2+\gamma_i^2(z-z_i)^2]} \end{aligned}$$

where $T(z)$ is given by

$$T(z) = \begin{cases} T_0 & \text{if } z \leq z_1 \\ T(z_1) + \dfrac{dT}{dz_1}(z - z_1) & \text{if } z_1 < z \leq z_2 \\ T(z_2) + \dfrac{dT}{dz_2}(z - z_2) & \text{if } z > z_2 \end{cases}$$

The coefficients α_i, β_i and γ_i should be as high as less spreaded is the spot.

8.3 Condutivity

The condutivity sensor emulator works just like the temperature one. However, the standard condutivity profile is the inverse to the one of the temperature, so the values of the derivatives will be positive.

8.4 Echosounder

The Echosounder allows the vehicle to know its distance to the bottom. It can be emulated in a similar way as the pressure sensor emulator, that is, we can start with a base distance to bottom and then apply some coefficients for the coordinates on the xx and yy axis. We have

$$D(x, y) = D_0 + D_1 x + D_2 y$$

In order to have a bottom profile, with elevations or depressions, we may add

$$\sum_i h_i e^{-[\alpha_i^2(x-x_i)^2+\beta_i^2(y-y_i)^2]}$$

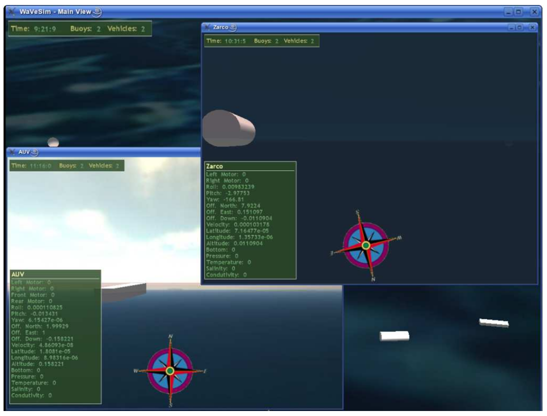
to the above value. We don't need the spread over the zz axis since it is already limited by h_i. It must be mentioned that this gives the distance to the bottom measured from water surface. To have the measure from vehicle's position, it is only needed to subtract the vehicle's depth.

8.5 Salinity

The value for the salinity is estimated using rather complex formulas that take the measures for temperature, condutivity and depth as parameters. The explanation of that formulas is out of this article's scope but can be checked in (Fofonoff and Millard, 1983).

9. *WAVESIM3D*

One of the first requirements when developing *WaVeSim* was to establish a complete separation between the simulator itself and any visualization system that would possibly be developed. This allows the simulation to run with any visualization system or with none at all. This requirement led to the idea of storing all the simulation data on shared memory areas. For example, this allows a web-based visualization through dynamic pages or with an applet. However, here we introduce a 3D visualization system that was developed alongside the simulator. This system was built using OGRE (Object-Oriented Graphics Rendering Engine) (OGRE, 2006), an open-source C++ 3D API which, in this case, uses the OpenGL rendering system. *WaVeSim3D* shows a main visualization window with a free camera that can be controlled with the mouse and keyboard and from which the user can interact with *WaVeSim* through some keyboard shortcuts. The simulation elapsed time, the number of buoys and the number of vehicles are also displayed. *FrameListeners* are classes that have methods that are called before and after each frame is rendered. *WaVeSim3D* uses one *FrameListener* to update all the clients' 3D entities frame by frame. These entities are added/removed to the scene as each client enters/exits the simulation. This class gets each body's transformation matrix from *WaVeSim*'s clients' shared memory area, sets the 3D entity's orientation by extracting a quaternion from the 3×3 rotation sub-matrix and sets the entity's position by extracting the translation part from the transformation matrix. Usually, the axis standard in 3D APIs is different from the North-East-Down standard used in the navigation area, so first the transformation matrix must be transposed. To see each vehicle's details, it is possible to click on it on the main render window. A new render window will open with a view from a fixed camera on the front the vehicle. Also, an overlay with the vehicle's data is displayed on top. One can check almost all the properties mentioned in 5. Also, a



```
Time: 9:21:9     Buoys: 2  Vehicles: 2
```

```
Time: 10:31:5    Buoys: 2  Vehicles: 2
Zarco
Left Motor: 0
Right Motor: 0
Roll: 0.00983239
Pitch: -2.97753
Yaw: -166.81
Off. North: 7.9224
Off. East: 0.151097
Off. Down: -0.0110904
Velocity: 0.000103178
Latitude: 7.16477e-05
Longitude: 1.35733e-06
Altitude: 0.0110904
Bottom: 0
Pressure: 0
Temperature: 0
Salinity: 0
Condutivity: 0
```

```
Time: 11:16:0    Buoys: 2  Vehicles: 2
AUV
Left Motor: 0
Right Motor: 0
Front Motor: 0
Rear Motor: 0
Roll: 0.000110825
Pitch: -0.013431
Yaw: 6.15427e-06
Off. North: 1.99929
Off. East: 1
Off. Down: -0.158221
Velocity: 4.86093e-08
Latitude: 1.8081e-05
Longitude: 8.98316e-06
Altitude: 0.158221
Bottom: 0
Pressure: 0
Temperature: 0
Salinity: 0
Condutivity: 0
```

Fig. 3. *WaVeSim3D*

compass image is drawn on each window so the user can have a precise indication to where each camera is pointing to.

10. *WAVESIM* LIBRARY

To allow any external process to access *WaVeSim* data or to allow any client to connect to the simulator, a small SDK is provided to ease the development of these components. The API provides classes to access the shared memory areas and info about the commander message queue. Also, the main data structures are also defined in some include files. To allow an easy development of a broker application, there's also a couple of classes that establish the connection to *WaVeSim* and provide some methods for most of the interactions so that the user shouldn't deal with the messaging protocol details. The incoming messages from *WaVeSim* are dropped in a user named POSIX message queue and the application is notified of new messages by an user-defined system signal.

11. CONCLUSION

Despite *WaVeSim* still being under development (including the external components like the sensor emulators and *WaVeSim3D*), it is already quite stable and fully operational. In fact, it is already being used to help the development of the Ocean Systems Group vehicles' on-board software. The use of this simulator by other entities to test their own control and navigation software will assert the usefulness of this application and will allow to extend it in order to support other demanded features based on gathered user experience.

ACKNOWLEDGEMENTS

This work was supported by the project MUV — *Multiple Underwater Vehicles* — funded by FCT and Programa POSI (POSI/SRI/47351/2002) and by the project INCORP - *Improved Navigation with Cooperative Robotic Platforms* - funded by FCT and Programa POSC (POSC/EEA-SRI/59963/2004).

REFERENCES

Braünl, Thomas (2006). Subsim - an autonomous submarine simulation system. **URL:** *http://robotics.ee.uwa.edu.au/auv/subsim.html.*

Coates, R. (2001). The sonar equations. In: *The Sonar Course.*

Fofonoff, N. P. and R. C. Millard (1983). Algorithms for computation of fundamental properties of seawater. *UNESCO Technical Papers in Marine Science, 44.*

Fossen, Thor Inge (1991). Nonlinear Modelling and Control of Underwater Vehicles. PhD thesis. Norwegian Institute of Technology.

Jerez, Julio (2006). Newton game dynamics. **URL:** *http://www.newtondynamics.com.*

NPS (2006). Nps auv workbench. **URL:** *http://terra.cs.nps.navy.mil/AUV/workbench/install.htm.*

OGRE (2006). Ogre 3d. **URL:** *http://www.ogre3d.org.*