Stonefish Documentation

Release 1.1.0

Patryk Cieślak

Sep 23, 2020

Contents

CHAPTER 1

An advanced simulation tool developed for marine robotics

Stonefish is a C++ library combining a physics engine and a lightweight rendering pipeline. The physics engine is based on the core functionality of the Bullet Physics library, extended to deliver realistic simulation of marine robots. It is directed towards researchers in the field of marine robotics but can as well be used as a general purpose robot simulator.

Stonefish includes advanced hydrodynamics computation based on actual geometry of bodies to better approximate hydrodynamic forces and allow for effects not possible when using symbolic models. The rendering pipeline, developed from the ground up, delivers realistic rendering of atmosphere, ocean and underwater environment. Special focus was put on the latter, where effects of wavelength-dependent light absorption and scattering were considered.

Stonefish can be used to create standalone simulators or combined with a ROS package stonefish_ros, which implements a standard simulator node, loading the simulation world from an XML description file. This simulator node takes care of ROS interaction through messages and services. The XML parser function can as well be used in the custom standalone simulator.

CHAPTER 2

Cite Me

This software was written and is continuously developed by Patryk Cieślak. Parts of the software based on code developed by other authors are clearly marked as such.

If you find this software useful in your research, please cite:

Patryk Cieślak, "Stonefish: An Advanced Open-Source Simulation Tool Designed for Marine Robotics, With a ROS Interface", In Proceedings of MTS/IEEE OCEANS 2019, June 2019, Marseille, France

```
@inproceedings{stonefish,
   author = "Cie{\'s}lak, Patryk",
   booktitle = "OCEANS 2019 - Marseille",
   title = "Stonefish: An Advanced Open-Source Simulation Tool Designed for Marine␣
↪Robotics, With a ROS Interface",
   month = "jun",
   year = "2019",
   doi = "10.1109/OCEANSE.2019.8867434"
}
```

CHAPTER 3

Funding

This work was part of a project titled ”Force/position control system to enable compliant manipulation from a floating I-AUV”, which received funding from the European Community H2020 Programme, under the Marie Skłodowska-Curie grant agreement no. 750063. The work was continued under a project titled ”EU Marine Robots”, which received funding from the European Community H2020 Programme, grant agreement no. 731103.

3.1 Overview

3.1.1 Features

In the following sections most importat features of the *Stonefish* library are presented. Features specifically implemented for the marine robotics research are written in **bold**. The full potential of the software can be appreciated by familiarizing with the subsequent chapters of the documentation.

Physics and collision

The *Stonefish* library is used to build a dynamic simulation world with rigid body physics, **geometry-based hydrodynamics** and impulse-based collisions. Rigid body dynamics and collision simulation are supplied by the *Bullet Physics* library, while hydrodynamics and custom material models are implemented by the author.

Kinematic trees of rigid bodies are supported and solved using the Featherstone’s algorithm. Collision is detected analytically between simple solids, using convex hull algorithms for dynamic meshes and concave-convex algorithms for static meshes. Material properties include density and coefficient of restitution, while interaction between different materials is described by a table of static and dynamic friction coefficients. Moreover, physical properties of bodies - mass, volume, inertia - are automaticall computed based on the supplied geometry and defined material.

Hydrodynamic and hydrostatic forces are approximated taking into account the actual geometry of the bodies, to deliver effects not possible when using analytic equations. These forces include **buoyancy, different types of drag (linear, quadratic and skin) and added mass**. Thanks to using real geometry in computations, buoyancy of partially submerged bodies can be reallisticaly simulated, as well as, local water velocity is impacting the behaviour of the bodies.

Actuators, sensors and communication devices

Apart from the pure physics simulation, the *Stonefish* library delivers a wide spectrum of virtual actuators, sensors and communication devices.

Actuators is a group of devices that can propel a body by generating a force attached to it or drive joints of a multi-body chain. These include **thrusters** and propellers, servos and motors and a **variable buoyancy system (VBS)**. Lights are also considered actuators.

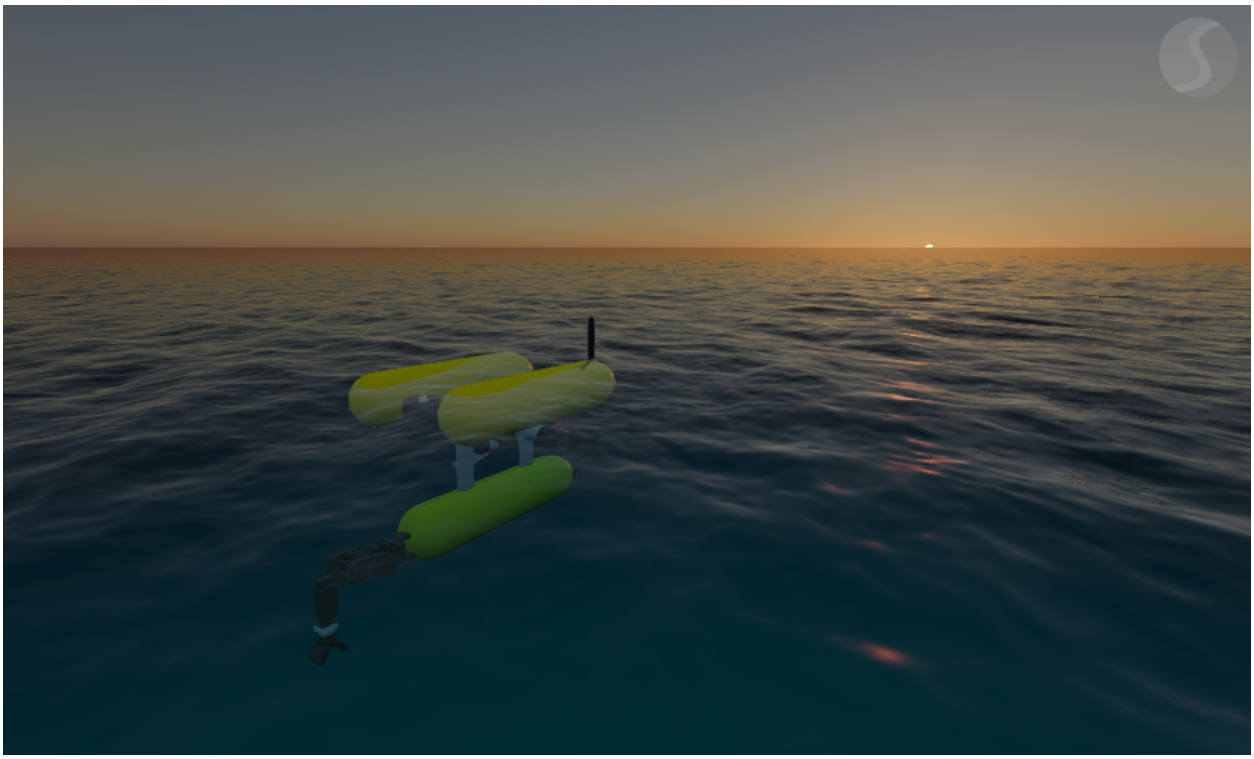
Sensors is a large group of different types of devices that can measure the interal states of the multi-body joints, the motion parameters of the bodies as well as environmental quantities. They can be divided into joint, link and vision sensors. Joint sensors is the smallest group, containing encoders, torque and force sensors. Link sensors is the biggest group including: IMU, odometry sensors, **GPS, compass, pressure sensor, Doppler velocity log (DVL), profiler and multi-beam sonar**. Finally, vision sensors include all devices that generate an image like: color camera, depth camera, **forward-looking sonar (FLS), mechanical scanning imaging sonar (MSIS) and side-scan sonnar (SSS)**.

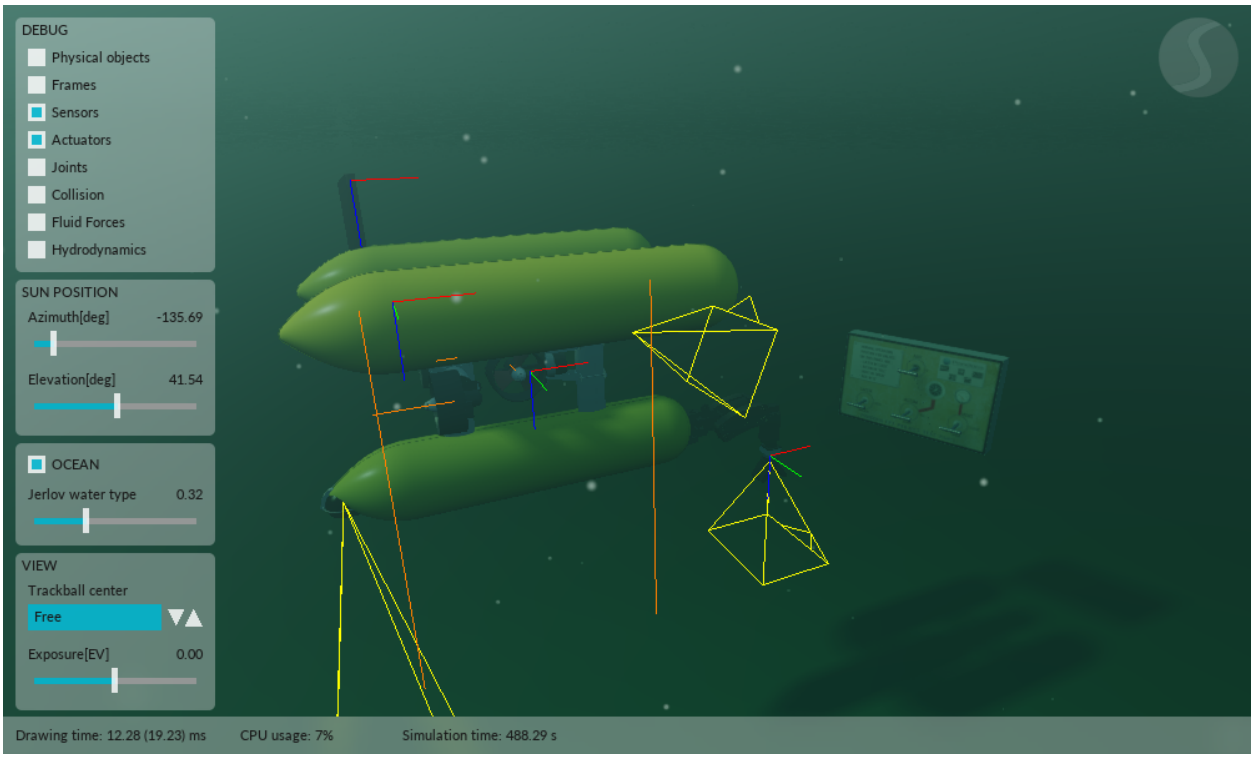
Comminication devices is the youngest group of simulated devices and it was created mainly to account for delays and directivity of communication underwater. The implemented communication devices include: **an acoutic modem and the ultra-short base line (USBL) device**.

Reallistic rendering

Another unique feature of the *Stonefish* library is its custom rendering pipeline, closely coupled with the physics engine. The reasons of not using any of the available 3D graphics engines was to avoid bulkiness and deliver **high quality sky, ocean and underwater rendering**, often missing in the available open-source solutions. Special focus was put on the simulation of the underwater environment, where effects of **wavelength-dependent light absorption and scattering based on Jerlov measurements** were considered.

3.1.2 Screenshots





3.2 Theory

3.2.1 Unit system

The International System of Units (SI) is used for the definitions and computations throughout the simulation library. For convenience some of the angular quantities are specified in degrees; the computations are always done using radians.

3.2.2 Coordinate frames

Multiple coordinate frames are defined in marine craft theory. At this stage, the *Stonefish* library is intended for small scale simulations where Earth curvature can be neglected. Therefore, the **North-East-Down (NED) coordinate frame** is used throughout the simulation, as the world reference frame. The NED frame is a Cartesian coordinate frame, tangent to the Earth surface at a chosen geographic location. This location is called the home and defined by latitude and longitude.

3.2.3 Rigid body dynamics and collision

The *Stonefish* library utilises algoritms implemented in the *Bullet Physics* library for the computation of the rigid body kinematics, dynamics and collision. The simulation engine is implementing velocity-based dynamics with semi-implicit Euler integration. The dynamics are computed using the Sequential Impulse algorithm for separate dynamical bodies. Kinematic trees are handled through an implementation of the Featherstone multi-body algorithm. Rigid collisions are computed using an impulse-based approach. Soft collisions are possible by defining collision stiffeness and damping factors. Frictional forces are computed based on static and dynamic friction coefficients defined between materials. The Stribeck function is used for the transition between sticking and sliding phases.

3.2.4 Hydrodynamics

The *Stonefish* library delivers a novel approach to simulating hydrodynamics and aerodynamics, by performing geometry based computations. The main focus is put on hydrodynamics as the library is directed towards marine robotics. The simulated effects include: added mass, buoyancy and drag. The drag is composed of 3 elements: potential drag (linear), form drag (quadratic) and skin drag.

Added mass

The added mass effect is encountered during the acceleration of bodies submerged in liquid. It is manifested by the dynamical response of the body as if it was heavier than it actually is, because the surrounding liquid has to be moved together with it. The added mass is formulated as a 6x6 matrix, which describes how acceleration in every direction is affected by the fluid. It is a common practice to use a diagonal representation of the matrix, which is also the case here. Moreover, due to the fact that the underlying physics library is not capable of accepting different values of mass for each axis, a mean value is used for all of the axes. In terms of the added inertia tensor, a vector of 3 inertia moments is used. The values of the added mass matrix are computed based on an automatic approximation of the body geometry using one of the 3 solids: sphere, cylinder or ellipsoid.

Buoyancy

The buoyancy force is calculated based on the sum of hydrostatic forces acting on the body surface. The actual geometry is used to compute force at each face of the mesh, depending on the depth of the face centre. It allows for simulating realistic buoyancy force at the surface of the ocean, with and without geometrical waves. When the body is completely submerged the buoyancy force is based on the volume of the mesh, computed automatically during loading.

Drag

The drag forces are calculated as a sum of forces acting on each face of the body surface. To obtain precise values of these forces it is required to solve Navier-Stokes equations, which is not possible for a general 3D case in realtime. Therefore, the computations implemented in the *Stonefish* library have to be based on the local velocity of fluid as if there was no body. The result is not quantitively correct but it gives a good approximation and allows for effects not possible when using simple formulas, e.g., a water current acting on a part of the body.

3.3 Installation

3.3.1 Requirements

The *Stonefish* library requires a modern multi-core processor to run the physics calculations in realtime.

A discrete graphics card with the support for **OpenGL 4.3 or higher** is required to run graphical simulations.

3.3.2 Dependencies

The following dependencies have to be installed prior to building the library:

- OpenGL Mathematics (libglm-dev, version >= 0.9.9.0)
- SDL2 (libsdl2-dev)

Note: SDL2 library may need a small fix to the cmake configuration file, to avoid build errors. Remove a space after `-lSDL2` in `/usr/lib/x86_64-linux-gnu/cmake/SDL2/sdl2-config.cmake`.

- Freetype (libfreetype6-dev)

3.3.3 Building

The following steps are needed to build the library:

```
$ git clone "https://github.com/patrykcieslak/stonefish.git"
$ cd stonefish
$ mkdir build
$ cd build
$ cmake ..
$ make -jX
$ sudo make install
```

3.3.4 Generating code documentation

The following steps are needed to generate and open the documentation of the library code.

1. Go to "stonefish" directory.
2. `$ doxygen doxygen`
3. Open "docs/html/index.html".

3.4 Building a simulator

3.4.1 Types of simulators

The *Stonefish* library is designed to build simulators for specific scenarios, by subclassing a minimal number of classes and overriding as few methods as possible. Depending on the functionality that is requested it can be as little as one class and one method. Moreover, there are two different kinds of simulators that can be built: a *console mode* simulator and a *graphical mode* simulator. A *console mode* simulator does not provide any functionality that requires graphics, which includes not only visualisation of the simulated scenario but also simulation of cameras, lights, depth map based sensors and waves. This kind of simulators can run on platforms which do not conform to the minimum requirements of the rendering pipeline. The normal mode of operation of the simulators is graphical.

Note: The *Stonefish* repository contains examples of different types of simulators in the `Tests` directory. These examples range from simple dynamic bodies falling on a groud plane to a full model of an autonomous underwater vehicle (AUV).

3.4.2 Building a simple graphical simulator

The following steps have to be completed to create a simple graphical simulator:

1. Create a new class `MySimulationManager` being a subclass of `sf::SimulationManager`.

2. Override method `void BuildScenario()` of the base class, **creating the whole simulation scenario inside**.
3. Create a .cpp file with the `int main(int argc, char **argv)` function.
4. Include the header files: `#include <Stonefish/core/GraphicalSimulationApp.h>` and `#include "MySimulationManager.h"`.
5. Create and fill two structures of types `sf::RenderSettings` and `sf::HelperSettings`.
6. Create a new object of class `MySimulationManager`.
7. Create a new object of class `sf::GraphicalSimulationApp`, passing the previous one to the constructor, together with the structures defined before.
8. Use method `void Run()` from the second object.
9. Build application and link with libraries.
10. Run the simulator from the terminal.

Following the steps above should result in 3 new files: *main.cpp*, *MySimulationManager.h* and *MySimulationManager.cpp*. The exemplary contents of these files are attached below. The simulated scenario is a ball bouncing on a flat surface.

main.cpp:

```
#include <Stonefish/core/GraphicalSimulationApp.h>
#include "MySimulationManager.h"

int main(int argc, char **argv)
{
    //Using default settings
    sf::RenderSettings s;
    sf::HelperSettings h;

    MySimulationManager manager(500.0);
    sf::GraphicalSimulationApp app("Simple simulator", "path_to_data", s, h, &
↪manager);
    app.Run();

    return 0;
}
```

MySimulationManager.h:

```
#include <Stonefish/core/SimulationManager.h>

class MySimulationManager : public sf::SimulationManager
{
public:
    MySimulationManager(sf::Scalar stepsPerSecond);
    void BuildScenario();
};
```

MySimulationManager.cpp

```
#include "MySimulationManager.h"
#include <Stonefish/entities/statics/Plane.h>
#include <Stonefish/entities/statics/Sphere.h>
```

(continues on next page)

(continued from previous page)

```
MySimulationManager::MySimulationManager(sf::Scalar stepsPerSecond) :␣
↪SimulationManager(stepsPerSecond)
{
}

void MySimulationManager::BuildScenario()
{
    //Physical materials
    CreateMaterial("Aluminium", 2700.0, 0.8);
    CreateMaterial("Steel", 7810.0, 0.9);
    SetMaterialsInteraction("Aluminium", "Aluminium", 0.7, 0.5);
    SetMaterialsInteraction("Steel", "Steel", 0.4, 0.2);
    SetMaterialsInteraction("Aluminium", "Steel", 0.6, 0.4);

    //Graphical materials (looks)
    CreateLook("gray", sf::Color::Gray(0.5f), 0.3f, 0.2f);
    CreateLook("red", sf::Color::RGB(1.f,0.f,0.f), 0.1f, 0.f);

    //Create environment
    sf::Plane* plane = new sf::Plane("Ground", 10000.0, "Steel", "gray");
    AddStaticEntity(plane, sf::I4());

    //Create object
    sf::Sphere* sph = new sf::Sphere("Sphere", 0.1, sf::I4(), "Aluminium",␣
↪sf::BodyPhysicsType::SURFACE_BODY, "red");
    AddSolidEntity(sph, sf::Transform(sf::IQ(), sf::Vector3(0.0,0.0,-1.0)));
}
```

3.4.3 Interacting with the simulator

Physical controllers

The *graphical mode* simulators can be interacted with using physical controllers - keyboard and/or joystick/gamepad. To implement this physical interaction it is necessary to subclass `sf::GraphicalSimulationApp`. Some of the base class methods need to be overridden. When keyboard is to be used as a controller the methods to override are called `void KeyDown(SDL_Event* event)` and `void KeyUp(SDL_Event* event)`. Care has to be taken when overriding the `KeyDown()` method, because its base class version implements some default keyboard functionality. If this functionality is to be retained the base class method has to be called in the subclass. If the joystick/gamepad support is to be implemented, the methods to override are called `void JoystickDown(SDL_Event* event)`, `void JoystickUp(SDL_Event* event)` and `void ProcessInputs()`. The former two are used to implement joystick button functionality while the latter can be used to read values of the joystick axes.

Note: If the standard keyboard handling was not overridden, the `w s a d z x` keys can be used to manipulate the viewport. The mouse can be used to select objects (left button), rotate the viewport (right button) and move the rotation centre (middle button).

Internal or external code

Any type of simulator will probably require some interaction with internal or external code. This can be a control algorithm implemented inside the simulator application or another application that requests data from the

simulator, like sensor readings, and/or wants to modify actuator setpoints. To ensure consistency of the simulation results this data can only be read and written at specific moments in time. To facilitate easy interaction the class `sf::SimulationManager` provides a virtual method `void SimulationStepCompleted(Scalar timeStep)`, which is called by the physics engine after a single simulation step is completed. Since the base class has to be subclassed to build a simulation scenario, it is easy to override another method for the interaction purposes.

Robot Operating System (ROS)

One of the anticipated uses of the simulators built with the *Stonefish* library, is their use in combination with ROS. The simulator can substitute the real robot during the research and development phases, transparently hooking up to the complex, ROS-based, control and navigation architectures. The interfacing is done using the same basic idea of overriding the `void SimulationStepCompleted(Scalar timeStep)` method. The author provides a ROS package called stonefish_ros, delivering a standard simulator node, to simplify the integration.

3.4.4 Graphical user interface (GUI)

The *Stonefish* library uses the concept of an immediate-mode GUI (IMGUI), which is displayed in the simulation window. It is a non-retained user interface, which means that the programmer is responsible for all data management, and the GUI is always rendered based on the current data. The IMGUI offers a few basic widgets to enable displaying and manipulating simulation parameters as well as plotting sensor measurements. There is a standard implementation of the IMGUI, displayed on the left side of the window, which can be overwriten or enhanced. Apart from the standard widgets, a text console is implemented which is always displayed during the simulator startup and can be used to check for errors and warnings.

Note: If the standard keyboard handling was not overridden, the `h` key can be used to show/hide the IMGUI and the `c` key can be used to show/hide the console. The console can be scrolled using the mouse.

Customising the IMGUI

To customise the IMGUI, the class `sf::GraphicalSimulationApp` has to be subclassed and the method `void DoHUD()` has to be overridden. Every widget has to use a unique `sf::ui_id`, which allows for identification of active IMGUI elements.

The available widgets include:

- `Panel` a box that may be used to group items visually.
- `Label` a static text
- `ProgressBar` an indicator which shows progress in form of a partially filled rectangle
- `Button` a momentary button that can be pressed
- `Slider` slider that can be used to set a continuous parameter
- `CheckBox` a checkbox that can be used to enable/disable options
- `ComboBox` a box with multiple selectable options
- `TimePlot` a plot which can display data from one or more sensor channels, with a common time axis
- `XYPlot` a plot which can display a relation between two sensor channels

Example of creating a button widget (*note:* `MySimulationApp` is a subclass of `sf::GraphicalSimulationApp`):

```
void MySimulationApp::DoHUD()
{
    GraphicalSimulationApp::DoHUD(); //Keep standard GUI

    sf::ui_id button;
    button.owner = 1; //e.g. id of a panel
    button.index = 0; //e.g. id of a widget on the panel
    button.item = 0; //e.q. id of an option on a list

    if(getGUI()->DoButton(button, 200, 10, 200, 50, "Press me"))
        code_to_execute;
}
```

3.5 Defining a simulation scenario

The simulation scenario can be defined in two ways: by writing code or by using a scenario file parser. Both ways can also be combined, by first loading the scenario from a file and then modifying it using code. This gives access to the full functionality of the library, allows for non-standard definitions, as well as opens the possibility of using extensions developed for the library. In all cases, the definitions have to be placed inside the `void BuildScenario()` method of your subclass of `sf::SimulationManager`.

3.5.1 Using the scenario file parser

For the convenience of the users, the *Stonefish* library implements a parser class for loading simulation scenarios from specially formulated scenario files. It is the preferred method of setting up the simulation scenarios, without the need of writing code. Loading a scenario from a file can be achieved through the following lines of code:

```
sf::ScenarioParser parser(this);
parser.Parse("path_to_scenario_file");
```

Scenario file syntax

Scenario files are XML files, with a syntax similar to the URDF. Due to the unique features of the *Stonefish* library, some of the tags used in the scenario files are specific to it, and different from other formats. Every scenario file has to contain a root node called `<scenario>`, i.e., all definitions have to be written between the tags `<scenario> ... </scenario>`. All paths defined in the scenario files are automatically recognised as absolute if they begin with `/` or `~`, or as relative to the data directory passed to the constructor of the `sf::SimulationApp` otherwise. Whenever an attribute requires passing multiple values, these values have to be separated by spaces, e.g., `<world_transform rpy="0.0 3.1415 0.0" xyz="1.0 2.0 3.0"/>`.

A properly defined scenario file has to contain a set of **obligatory tags**, specifying the type of the simulated environment as well as the list of the materials and looks used throguhout the scenario. The rest of the definitions are used to actually create the simulated static bodies, dynamics bodies and robots. A skeleton of a scenario file is shown below:

```
<?xml version="1.0"?>
<scenario>
    <environment>
        <!-- Parameters of simulated environment -->
    </environment>
    <materials>
        <!-- Definitions of materials -->
```

(continues on next page)

(continued from previous page)

```
        <friction_table>
            <!-- Interaction between materials -->
        </friction_table>
    </materials>
    <looks>
        <!-- Definitions of looks -->
    </looks>
    <!-- Definitions of bodies and robots -->
</scenario>
```

It is possible to **include one file in another**, to allow for the reuse of common definitions. The includes can only be used at the root level. The include mechanism also supports passing arguments that are automatically replaced when loading the file contents. To define an argument the user has to add `<arg name="{1}" value="{2}"/>` to the `<include>` node, where {1} is the argument name and {2} is the argument value. In the included file the argument is retrieved by using `$(arg {1})` inside an attribute value. An example of including a file with arguments is presented below:

```
<!-- main.scn -->
<?xml version="1.0"?>
<scenario>
    <!-- some definitions -->
    <include file="robot.scn">
        <arg name="robot_name" value="GIRONA500"/>
    </include>
</scenario>

<!-- robot.scn -->
<?xml version="1.0"?>
<scenario>
    <robot name="$(arg robot_name)" fixed="false">
        <!-- robot definitions -->
    </robot>
</scenario>
```

3.5.2 Using the code

Using the code to create simulation scenarios has a few possible benefits. First of all, it is necessary in case of extending the functionality of the *Stonefish* library, e.g., with new sensors, actuators or communication devices. This necessity can be dropped if the parser class is extended to include this new functionality. The other option is to load the standard definitions from a scenario file and add the missing elements with code. Secondly, the library code might expose properties and functions not supported by the parser, which may happen due to the difficulty in defining a particular functionality through the scenario file syntax. Finally, using code allows for implementing dynamically created simulation worlds, possibly with parametric functions, random distribution of bodies, generated terrain, etc.

When no scenario file is used, all of the obligatory definitions have to be written with code, in a specific order. Naturally, the materials and looks have to be defined before they can be used, which is not the case with the scenario file, in which the order of the tags does not matter. Moreover, the programmer is fully responsible for the correctness of the defined scenario, as any error checking mechanisms, implemented inside the parser, are not working anymore.

Note: The rest of the documentation describes in detail how to define the obligatory properties of a simulation world, as well as every implemented object, that can be used in a simulation scenario. Each of the descriptions is accompanied by an XML snippet and its C++ twin, showing how to create objects using the scenario file syntax or the code.

3.6 Materials

Before the environment and the robots can be created, two groups of materials have to be defined in the simulation scenario. These groups are: the physical materials, defining the physical properties of the bodies, and the graphical materials (looks), defining how the bodies will look. The physical materials and the looks are defined separately to allow for arbitrary mixing between them, when creating the bodies. Moreover, bodies made of the same physical material may require distinctive looks.

Note: The names of the defined physical materials and looks have to be used when creating the objects in the simulation scenario (case-sensitive). Therefore, the materials have to be defined prior to the definition of any bodies.

3.6.1 Physical materials

Physical materials are characterised by physical properties: the density and the restitution factor. This allows for automatic calculation of mass and inertia of the bodies, based on their geometry. The restitution factor defines the fraction of momentum preserved during rigid collisions. It also affects the way that the body reflects the sonar waves, as they are mechanical in nature. There is also a mechanism to define frictional interaction between different materials, by setting static and dynamic friction coefficients between all possible material pairs.

An example of the definition of two physical materials, using the XML syntax, is presented below:

```
<materials>
    <material name="Aluminium" density="2710.0" restitution="0.7"/>
    <material name="Steel" density="7891.0" restitution="0.9"/>
    <friction_table>
        <friction material1="Aluminium" material2="Aluminium" static="0.7" dynamic="0.
↪4"/>
        <friction material1="Steel" material2="Steel" static="0.4" dynamic="0.1"/>
        <friction material1="Aluminium" material2="Steel" static="0.8" dynamic="0.5"/>
    </friction_table>
</materials>
```

The same can be achieved using the following code:

```
CreateMaterial("Aluminium", sf::Scalar(2791), sf::Scalar(0.7));
CreateMaterial("Steel", sf::Scalar(7891), sf::Scalar(0.9));
SetMaterialsInteraction("Aluminium", "Aluminium", sf::Scalar(0.7), sf::Scalar(0.4));
SetMaterialsInteraction("Steel", "Steel", sf::Scalar(0.4), sf::Scalar(0.1));
SetMaterialsInteraction("Aluminium", "Steel", sf::Scalar(0.8), sf::Scalar(0.5));
```

3.6.2 Looks

The graphical materials, called looks, define how the objects are rendered, thus they only exist in graphical mode simulations. All looks are rendered with a Cook-Torrance model, parametrised by reflectance (color), roughness, metalness and reflection factor (0.0 = no reflections, 1.0 = mirror). Additionally, it is possible to attach two textures to a material: an albedo texture and a normal texture. The albedo texture is multiplied by the reflectance to define the color of the material. The normal texture (map) is used to introduce surface details which would otherwise require complex geometry (bump mapping). The normal vectors computed from the geometry are rotated locally, based on the normal map, before the lighting calculations are performed. The normal map also affects the normals seen by the sonar, which allows for capturing realistic surface textures in the virtual sonar images.

Example of defining different looks, using the XML syntax, is presented below:

```
<looks>
    <look name="Yellow" rgb="1.0 0.9 0.0" roughness="0.3"/>
    <look name="Gray" gray="0.3" roughness="0.4" metalness="0.5"/>
    <look name="Textured" gray="1.0" roughness="0.1" texture="tex.png"/>
    <look name="Bump" rgb="1.0 0.0 0.0" roughness="0.2" normal_map="normal.png"/>
</looks>
```

The same can be achieved using the following code:

```
CreateLook("Yellow", sf::Color::RGB(1.f, 0.9f, 0.f), 0.3f);
CreateLook("Gray", sf::Color::Gray(0.3f), 0.4f, 0.5f);
CreateLook("Textured", sf::Color::Gray(1.f), 0.1f, 0.f, 0.f, sf::GetDataPath() + "tex.
↪png");
CreateLook("Bump", sf::Color::RGB(1.f, 0.f, 0.f), 0.2f, 0.f, 0.f, "",␣
↪sf::GetDataPath() + "normal.png");
```

Note: Function `std::string sf::GetDataPath()` returns a path to the directory storing simulation data, specified during the construction of the `sf::SimulationApp` object.

3.7 Environment

The description of a simulation world starts with the definition of the environment to be simulated. The *Stonefish* library is prepared to be used as a general robot simulator, which implements crucial elements for marine robotics. The standard simulation medium is air, which can be used for the ground and flying robots (preliminary support). If the user is interested in simulating marine robots, a virtual ocean has to be enabled. Next, the world can be filled with static rigid bodies, which may include terrain and structures fixed to the world frame. If some parts of the environment are considered dynamic, they have to be created as dynamic bodies or robots, described in the subsequent parts of the documentation.

The parameters of the environment are specified between `<environment> ... </environment>`, inside the root node of the XML file, and they include position of the world frame origin (NED origin), the ocean definitions and the atmosphere definitions, explained in the following sections. If no ocean definitions are present the ocean simulation is disabled.

The basic structure of the XML tags used to define the environment is:

```
<environment>
    <ned latitude="40.0" longitude="3.0"/> <!-- geographic coordinates of the NED␣
↪origin -->
    <!-- ocean definitions -->
    <!-- atmosphere definitions -->
</environment>
```

The same can be achieved in code by the following line:

```
getNED()->Init(40.0, 20.0, 0.0);
```

3.7.1 Ocean

The ocean simulation is one of crucial parts of the *Stonefish* library. Obviously, marine robots can not be simulated well without reallistic hydrodynamics, interactions with the ocean surface and underwater currents. Moreover, ocean optics-based visuals are an important feature when trying to reproduce images from submerged cameras.

Waves

The library implements an ocean surface simulation utilising the fast Fourier transform (FFT), following the ideas of Tessendorf. Multiple FFT layers are computed using a GPU-based algoritm, to simulate the spectrum of the ocean waves and transform it into the 3D space and time domain. Later, the GPU generated data can be used to simulate the interaction between the ocean water and the dynamic bodies. This interaction is still under development and should be disable if not needed. Therefore, there is two ways the ocean can be simulated: with geometrical waves or as a flat surface. The flat surface option is also better in terms of performance.

Currents

Water currents have a significant impact on the operation of underwater robots. Therefore, the *Stonefish* library implements some basic forms of water currents, treated as water velocity fields. Currently implemented types of water currents include:

- `Uniform` the same velocity in the whole ocean
- `Jet` a velocity distribution coming from an circular underwater outlet
- `Pipe` a velocity distrubution resambling a virtual pipe submerged in the ocean

Ocean optics

As mentioned before, underwater rendering plays an important role in realistic simulation of optical sensors. The *Stonefish* library implements optical effects encountered in ocean waters like light absorption, out-scattering, and in-scattering, also called the airlight. The absorption and scattering coefficeints are computed for three wavelengths, corresponding to the red, green and blue channels of the rendering pipeline, based on the Jerlov measurements covering wide spectrum of the coastal water types. The water quality is defined with a single parameter ranging from 0.0 to 1.0, where the lower limit corresponds to the Jerlov type I water and the upper limit represents the Jerlov type 9C water.

Definitions

The ocean definitions have to be placed inside the environment node of the scenario file, between `<ocean> ... </ocean>`. An example covering all of the implemented features is presented below:

```
<ocean>
    <water density="1031.0" jerlov="0.2"/>
    <waves height="0.0"/>
    <current type="uniform">
        <velocity xyz="1.0 0.0 0.0"/>
    </current>
    <current type="jet">
        <center xyz="0.0 0.0 3.0"/>
        <outlet radius="0.2"/>
        <velocity xyz="0.0 2.0 0.0"/>
    </current>
</ocean>
```

The following lines of code can be used to achieve the same:

```
getMaterialManager()->CreateFluid("OceanWater", 1031.0, 0.002, 1.33);
EnableOcean(0.0, getMaterialManager()->getFluid("OceanWater"));
getOcean()->setWaterType(0.2);
```

(continues on next page)

(continued from previous page)

```
getOcean()->AddVelocityField(new sf::Uniform(sf::Vector3(1.0, 0.0, 0.0)));
getOcean()->AddVelocityField(new sf::Jet(sf::Vector3(0.0, 0.0, 3.0), sf::Vector3(0.0,␣
↪1.0, 0.0), 0.2, 2.0));
```

3.7.2 Atmosphere

The atmosphere simulation is another component of the virtual environment. It enables realistic motion of aerodynamic bodies, taking into account winds. In the current state only air drag is simulated. An important feature of the atmosphere simulation is the photo-realistic rendering of sky and Sun.

Winds

Winds have a significant impact on the motion of flying robots. Therefore, the *Stonefish* library implements some basic forms of wind, treated as air velocity fields. Currently implemented types of wind include:

- `Uniform` the same velocity in the whole atmosphere
- `Jet` a velocity distribution coming from an circular outlet
- `Pipe` a velocity distrubution resambling a virtual pipe submerged in the atmosphere

Sky and Sun

The sky and Sun rendering is based on precomputed atmospheric scattering algorithm. It takes into account multiple layers of Earth's atmosphere, including the ozone layer, to generate photo-realistic image of the sky. Sun's position on the sky can be changed dynamically during the simulation.

Definitions

The atmosphere definitions have to be placed inside the environment node of the scenario file, between `<atmosphere> ... </atmosphere>`. An example covering all of the implemented features is presented below:

```
<atmosphere>
    <sun azimuth="20.0" elevation="50.0"/>
    <wind type="uniform">
        <velocity xyz="1.0 0.0 0.0"/>
    </wind>
    <wind type="jet">
        <center xyz="0.0 0.0 3.0"/>
        <outlet radius="0.2"/>
        <velocity xyz="0.0 2.0 0.0"/>
    </wind>
</atmosphere>
```

The following lines of code can be used to achieve the same:

```
getAtmosphere()->SetupSunPosition(20.0, 50.0);
getAtmosphere()->AddVelocityField(new sf::Uniform(sf::Vector3(1.0, 0.0, 0.0)));
getAtmosphere()->AddVelocityField(new sf::Jet(sf::Vector3(0.0, 0.0, 3.0),␣
↪sf::Vector3(0.0, 1.0, 0.0), 0.2, 2.0));
```

3.7.3 Static bodies

The static bodies are all elements of the simulation scenario that remain fixed to the world origin, for the whole duration of the simulation. These kind of objects are used only for collision and sensor simulation. Due to their fixed position in the world, they do not require computation of dynamics and can deliver optimised collision detection algoritms. An important feature is that static bodies can have arbitrary collision geometry, not requiring convexity. Static bodies include a simple plane, basic solids, meshes and terrain. They are defined in the XML syntax using the `<static> ... </static>` tags, in a following way:

```
<static name="{1}" type="{2}">
    <!-- specific definitions here -->
    <material name="{3}"/>
    <look name="{4}"/>
    <world_transform xyz="{5a}" rpy="{5b}"/>
</static>
```

where

1) **Name**: unique string
2) **Type**: type of the static body
3) **Material name**: the name of the physical material
4) **Look name**: the name of the graphical material
5) **World transform**: position and orientation of the body with respect to the world frame.

Depending on the type of the static body the specific definitions change.

Note: In the following examples it is assumed that physical materials called "Steel" and "Rock", as well as looks called "Yellow" and "Gray", were defined.

Plane

A plane is the simplest static body, that is usually used as the ground plane or the sea bottom, if no complex terrain is needed.

In the XML syntax the plane does not have any additional parameters. It is only needed to define the type of the static body as "plane". An exemplary plane can be defined as follows:

```
<static name="Floor" type="plane">
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 1.0" rpy="0.0 0.0 0.0">
</static>
```

The same can be achieved in code:

```
sf::Plane* floor = new sf::Plane("Floor", 1000.f, "Steel", "Yellow");
AddStaticEntity(floor, sf::Transform(sf::Quaternion(0.0, 0.0, 0.0), sf::Vector3(0.0,␣
↪0.0, 1.0));
```

Note: Plane definition has one special functionality. It is possible to scale the automatically generated texture coordinates, to tile the textures associated with the look. In the XML syntax the `<look>` tag has to be augmented to

include attribute `uv_scale="#.#"` and in the C++ code the scale can be passed as the last argument in the object constructor.

Obstacles

The obstacles are static solids, created using parameteric definitions: spheres, cylinders and boxes, or loaded from geometry files.

In case of the **parameteric solids** the specific definitions are reduced to their dimensions. Both the physical and the graphical mesh have the same complexity. Depending on the shape a different set of dimensions has to be specified:

- **Sphere** `type="sphere"` - ball with a specified radius {1}:

```
<dimensions radius="{1}"/>
```

- **Cylinder** `type="cylinder"` - cylinder along Z axis, with a specified radius {1} and height {2}:

```
<dimensions radius="{1}" height="{2}"/>
```

- **Box** `type="box"` - box with specified width {1}, length {2} and height {3}:

```
<dimensions xyz="{1} {2} {3}"/>
```

Note: Box definition has one special functionality. It is possible to choose from 3 automatically generated texture coordinate schemes: scheme 0 (default) assumes that the texture is in a cubemap format and applies it to the box faces accordingly, scheme 1 applies the whole texture to each face of the box, and scheme 2 tiles the whole texture along each of the box faces, based on face dimensions. In the XML syntax the `<look>` tag has to be augmented to include attribute `uv_mode="#"` and in the C++ code the mode can be passed as the last argument in the object constructor.

Definition of arbitrary **triangle meshes** `type="model"`, loaded from geometry files, is more complex. Their geometry can be specified separately for the physics computations `<physical> .. </physical>` and the rendering `<visual> ... </visual>`. The physics mesh should be optimised to improve collision performance. If only physics geometry is specified, it is also used for rendering. Moreover, the physics mesh is used when simulating operation of *link sensors* and the graphics mesh is used for the *vision sensors*. The geometry can be loaded from STL or OBJ files (ASCII format).

An example of creating obstacles, including triangle meshes, is presented below:

```
<static name="Ball" type="sphere">
    <dimensions radius="0.5"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="2.0 0.0 5.0" rpy="0.0 0.0 0.0"/>
</static>

<static name="Wall" type="box">
    <dimensions xyz="10.0 0.2 5.0"/>
    <material name="Steel"/>
    <look name="Gray"/>
    <world_transform xyz="0.0 5.0 2.0" rpy="0.0 0.0 0.0"/>
</static>

<static name="Canyon" type="model">
    <physical>
```

(continues on next page)

(continued from previous page)

```
        <mesh filename="canyon_phy.obj" scale="1.0"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    </physical>
    <visual>
        <mesh filename="canyon_vis.obj" scale="1.0"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    </visual>
    <material name="Rock"/>
    <look name="Gray"/>
    <world_transform xyz="0.0 0.0 10.0" rpy="0.0 0.0 0.0"/>
</static>
```

The <origin> tag is used to apply local transformation to the geometry, i.e., to change the position of the mesh origin and rotate the mesh, before placing it in the world. It is especially useful in case of geometry exported from 3D software in a wrong frame.

The same can be achieved using the following code:

```
sf::Obstacle* ball = new sf::Obstacle("Ball", 0.5, "Steel", "Yellow");
AddStaticEntity(ball, sf::Transform(sf::Quaternion(0.0, 0.0, 0.0), sf::Vector3(2.0, 0.
↪0, 5.0)));
sf::Obstacle* wall = new sf::Obstacle("Wall", sf::Vector3(10.0, 0.2, 5.0), "Steel",
↪"Gray");
AddStaticEntity(wall, sf::Transform(sf::Quaternion(0.0, 0.0, 0.0), sf::Vector3(0.0, 5.
↪0, 2.0)));
sf::Obstacle* canyon = new sf::Obstacle("Canyon", sf::GetDataPath() + "canyon_vis.obj
↪", 1.0, sf::I4(), sf::GetDataPath() + "canyon_phy.obj", 1.0, sf::I4(), "Rock", "Gray
↪");
AddStaticEntity(canyon, sf::Transform(sf::Quaternion(0.0, 0.0, 0.0), sf::Vector3(0.0,␣
↪0.0, 10.0)));
```

Note: Function `std::string sf::GetDataPath()` returns a path to the directory storing simulation data, specified during the construction of the `sf::SimulationApp` object. Function `sf::Transform sf::I4()` creates an identity transformation matrix.

Terrain

Currently the *Stonefish* library implements one type of easily defined terrain mesh which is a heightmap based terrain `type="terrain"`. This kind of terrain mesh is generated from a planar grid displaced in the Z direction, based on the values of the heightmap pixels. Scale of the terrain is defined in meters per pixel and the height is defined by providing value correspondinng to a fully saturated pixel. The heightmap has to be a single channel (grayscale) image, with an 8 bit or 16 bit precision. The latter allows for much higher height resolution.

The following example presents the definition of a heightmap based terrain:

```
<static name="Bottom" type="terrain">
    <height_map filename="terrain.png"/>
    <dimensions scalex="0.1" scaley="0.2" height="10.0"/>
    <material name="Rock"/>
    <look name="Gray"/>
    <world_transform xyz="0.0 0.0 15.0" rpy="0.0 0.0 0.0"/>
</static>
```

```
sf::Terrain* bottom = new sf::Terrain("Bottom", sf::GetDataPath() + "terrain.png", 0.
↪1, 0.2, "Rock", "Gray");
AddStaticEntity(bottom, sf::Transform(sf::Quaternion(0.0, 0.0, 0.0), sf::Vector3(0.0,␣
↪0.0, 15.0)));
```

Note: Terrain definition has one special functionality. It is possible to scale the automatically generated texture coordinates, to tile the textures associated with the look. In the XML syntax the `<look>` tag has to be augmented to include attribute `uv_scale="#.#"` and in the C++ code the scale can be passed as the last argument in the object constructor.

3.8 Dynamic bodies

The dynamic bodies represent all of the rigid bodies that are not fixed to the world frame and thus require simulation of dynamics and hydrodynamics. It includes free bodies, constituting the dynamic parts of the *environment*, as well as *links of the robots*.

3.8.1 Common properties

All of the dynamic bodies share the same mechanical properties which are used in the computation of forces governing their motion. An important feature of the *Stonefish* library is that these properties are computed automatically, based on provided geometry and material definitions, and include: mass, moments of inertia, volume, location of the centre of gravity (**CG**) and location of the centre of buoyancy (**CB**).

Coordinate frames

The following diagram presents the coordinate frames defined for a dynamic body. The location of the frames is not realistic but only for illustration purposes. The shape of the body is described by: a graphical mesh (black), a mesh used for physics computations (green) and an approximation of the body geometry (dashed blue). There are 3 frames associated with these meshes: frame **G**, frame **C** and frame **H** respectively. The origin of the body is defined by frame **O**, which is used as a handle, to position the body in the world frame **W**.

Physics mode

Dynamic bodies are affected by different forces, depending on the type of environment, the position of the body with respect to the ocean surface (if it is enabled) and the selected body physics mode. The last one was introduced as an optimization to help determine which forces have to be computed for a specific body and thus how the body should be prepared for the simulation. The physics mode of each dynamic body `sf::BodyPhysicsType` has to be selected from one of the following options:

- `SURFACE` - no aerodynamic or hydrodynamic forces computed
- `FLOATING` - buoyancy and hydrodynamic drag is computed, no added mass effect
- `SUBMERGED` - buoyancy and hydrodynamic forces including added mass effect are computed
- `AERODYNAMIC` - aerodynamic drag is computed (lift not supported for general bodies)

Collisions

The *Stonefish* library uses a collision detection algorithm that approximates geometry of dynamic bodies to convex hulls. This feature significantly improves the performance of the simulation. Moreover, the library implements analytic collision points computation for basic solids, which should be used whenever possible. This not only further improves the performance, but also enables smooth collision response with standard curved surfaces. In case a non-convex collision is required, it is necessary to compose the dynamic body from multiple convex bodies, using the *compound body* type.

Creating dynamic bodies

The way of defining a dynamic body highly depends on its type. However, all of the dynamic bodies, except for the compound type, share some common properties:

1) **Name**: unique string
2) **Body type**: the type of the dynamic body
3) **Physics mode**: the physics computation mode
4) **Buoyant**: a flag indicating if the body is buoyant (optional)
5) **Material name**: the name of the material the body is made of
6) **Look name**: the name of the look used for rendering the body
7) **World transformation**: the transformation of the body origin in the world frame (position and orientation of the body)

```
<dynamic name="{1}" type="{2}" physics="{3}" buoyant="{4}">
    <!-- definitions specific for a selected body type -->
    <material name="{5}"/>
    <look name="{6}"/>
    <world_transform xyz="{7a}" rpy="{7b}"/>
</dynamic>
```

When creating the dynamic bodies in the C++ code, it is necessary to use a constructor of a specific body type. All of the dynamic body types are implemented as subclasses of `sf::SolidEntity`.

Note: In the following sections, description of each specific body type implementation is accompanied with an example of body instantiation through the XML syntax and the C++ code. It is assumed that a physical material called "Steel" and a look called "Yellow" were defined.

Overriding calculated properties

It is possible to override some of the automatically calculated properties of a dynamic body. There are two methods to do it:

1. Set an arbitrary mass and allow the library to automatically scale the moments of inertia

```
<dynamic>
    <!-- all standard definitions -->
    <mass value="30.0"/>
</dynamic>
```

```
sf::SolidEntity* solid = ...;
solid->ScalePhysicalPropertiesToArbitraryMass(30.0);
```

2. Set an arbitrary mass, moments of inertia and location of the CG

```
<dynamic>
    <!-- all standard definitions -->
    <mass value="30.0"/>
    <inertia xyz="1.0 0.5 0.2"/>
    <cg xyz="0.2 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
sf::SolidEntity* solid = ...;
solid->SetArbitraryPhysicalProperties(30.0, sf::Vector3(1.0, 0.5, 0.2),␣
↪sf::Transform(sf::IQ(), sf::Vector3(0.2, 0.0, 0.0)));
```

3.8.2 Parametric solids

The most efficient dynamic bodies are parametric solids, which include: box, sphere, cylinder, torus and wing. The physical geometry of parametric solids is the same as the graphical mesh. Besides the wing body, the collisions of parametric solids are computed analytically. Definition of parametric solids always includes the `dimensions` tag. The attributes of this tag depend on the type of the solid. There is one standard attribute which is always available, called `thickness`, used to define wall thickness if the user wants to create a shell body instead of a solid one.

1. Sphere `type="sphere"` - a sphere (ball) with a specified radius:

```
<dynamic name="Sphere" type="sphere" physics="submerged" buoyant="true">
    <dimensions radius="0.5"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
#include <Stonefish/entities/solids/Sphere.h>
sf::Sphere* sph = new sf::Sphere("Sphere", 0.5, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SUBMERGED, "Yellow");
AddSolidEntity(sph, sf::I4());
```

2. Cylinder `type="cylinder"` - a cylinder with a specified radius and height, with its axis coincident with the local Z axis:

```
<dynamic name="Cyl" type="cylinder" physics="surface">
    <dimensions radius="1.0" height="2.0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
#include <Stonefish/entities/solids/Cylinder.h>
sf::Cylinder* cyl = new sf::Cylinder("Cyl", 1.0, 2.0, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SURFACE, "Yellow");
AddSolidEntity(cyl, sf::I4());
```

3. Box `type="box"` - a box with specified width, height and length:

```
<dynamic name="Box" type="box" physics="submerged" buoyant="true">
    <dimensions xyz="0.5 1.0 2.0"/>
    <origin xyz="0.5 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 2.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
#include <Stonefish/entities/solids/Box.h>
sf::Box* box = new sf::Box("Box", sf::Vector3(0.5, 1.0, 2.0), sf::Transform(sf::IQ(),␣
↪sf::Vector3(0.5, 0.0, 0.0)), "Steel", sf::BodyPhysicsType::SUBMERGED, "Yellow");
AddSolidEntity(box, sf::Transform(sf::IQ(), sf::Vector3(0.0, 0.0, 2.0)));
```

4. Torus `type="torus"` - a torus with a specified major and minor radius, with its axis coincident with the local Y axis:

```
<dynamic name="Torus" type="torus" physics="submerged" buoyant="true">
    <dimensions major_radius="1.0" minor_radius="0.1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
#include <Stonefish/entities/solids/Torus.h>
sf::Torus* tr = new sf::Torus("Torus", 1.0, 0.1, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SUBMERGED, "Yellow");
AddSolidEntity(tr, sf::I4());
```

5. Wing profile `type="wing"` - a solid based on an extruded NACA profile (4-digit system), aligned with local Y axis:

```
<dynamic name="Wing" type="wing" physics="aerodynamic" buoyant="true">
    <dimensions base_chord="1.0" tip_chord="0.5" length="3.0" naca="4000"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

```
#include <Stonefish/entities/solids/Wing.h>
sf::Wing* wing = new sf::Wing("Wing", 1.0, 0.5, "4000", 3.0, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::AERODYNAMIC, "Yellow");
AddSolidEntity(wing, sf::I4());
```

3.8.3 Arbitrary meshes

The dynamic bodies can be created based on arbitrary geometry, loaded from mesh files `type="model"`. The geometry can be specified separately for the physics computation and the rendering. If only physical geometry is specified it is also used for rendering. The geometry can be loaded from STL or OBJ files (ASCII format).

```
<dynamic name="Mesh" type="model" physics="submerged" buoyant="true">
    <physical>
```

(continues on next page)

(continued from previous page)

```
        <mesh filename="model_phy.obj" scale="1.0"/>
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.0"/>
    </physical>
    <visual>
        <mesh filename="model_vis.obj" scale="1.0"/>
        <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.0"/>
    </visual>
    <material name="Steel"/>
    <look name="Yellow"/>
    <world_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

The <origin> tag is used to apply local transformation to the geometry, i.e., transformation in the frame defined by the 3D software used to save the geometry. Optionally, if the user wants to create a shell body instead of a solid body, a line <thickness value="#.#"/> has to be defined between the <physical> tags.

```
#include <Stonefish/entities/solids/Polyhedron.h>
sf::Polyhedron* poly = new sf::Polyhedron("Poly", sf::GetDataPath() + "model_vis.obj",
↪ 1.0, sf::I4(), sf::GetDataPath() + "model_phy.obj", 1.0, "Steel",␣
↪sf::BodyPhysicsType::SUBMERGED, "Yellow");
AddSolidEntity(poly, sf::I4());
```

3.8.4 Compound bodies

A special type of dynamic body, called *compound*, can be used, for intuitive construction of a group of rigidly connected elements and/or enabling correct collision with non-convex geometry. A compound body is composed of external and internal parts, with at least one obligatory external part. Only the external parts are used when computing the drag forces, while all parts contribute to the buoyancy. Each of the parts is defined as parametric or mesh body, using the previously presented syntax. The difference lies in how these bodies are added to the simulation world by first combining them into one compound body.

An example of creating a compound body is presented below:

```
<dynamic name="Comp" physics="submerged" type="compound">
    <external_part name="Part1" type="sphere" physics="submerged" buoyant="true">
        <dimensions radius="0.5"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <material name="Steel"/>
        <look name="Yellow"/>
        <compound_transform xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    </external_part>
    <internal_part name="Part2" type="box" physics="submerged" buoyant="true">
        <dimensions xyz="0.5 0.1 0.1"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <material name="Steel"/>
        <look name="Yellow"/>
        <compound_transform xyz="0.25 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <internal_part/>
    <world_transform xyz="0.0 0.0 5.0" rpy="0.0 0.0 0.0"/>
</dynamic>
```

It should be noticed that when defining parts of a compound body the <dynamic> tag is replaced with <external_part> and <internal_part> tags. The <compound_transform> tag defined for each of the parts is used to determine the position and orientation of the part in the origin frame of the compound body and it replaces the <world_transform>, which is now defined for the whole compound body, at the end.

```
#include <Stonefish/entities/solids/Sphere.h>
#include <Stonefish/entities/solids/Box.h>
#include <Stonefish/entities/solids/Compound.h>
sf::Sphere* part1 = new sf::Sphere("Part1", 0.5, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SUBMERGED, "Yellow");
sf::Box* part2 = new sf::Box("Part2", sf::Vector3(0.5, 0.1, 0.1), sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SUBMERGED, "Yellow");
sf::Compound* comp = new sf::Compound("Comp", part1, sf::I4(),␣
↪sf::BodyPhysicsType::SUBMERGED);
comp->AddInternalPart(part2, sf::Transform(sf::IQ(), sf::Vector3(0.25, 0.0, 0.0)));
AddSolidEntity(comp, sf::Transform(sf::IQ(), sf::Vector3(0.0, 0.0, 5.0)));
```

3.9 Robots

In the *Stonefish* library every kinematic tree is called a robot. The robots can thus represent vehicles, stationary manipulators, as well as other structures with moving parts. The kinematic tree is built of links (rigid bodies) connected by joints. When this mechanical structure is defined, then it is possible to attach *actuators*, *sensors* and *communication devices* to its joints and links. The dynamics of the tree are solved using the Featherstone's multi-body algorithm. Robots are defined using the `<robot>` tag. The base link of a robot can either be fixed to the world frame or floating. Robot definitions have to be placed in the root node of the XML file or after the definition of materials and looks in the C++ code.

3.9.1 Links

Links are the mechanical parts of the robot, joined together to create the kinematic tree. Links are considered *dynamic bodies* and are defined in the same way. To differentiate links from free dynamic bodies the XML syntax introduces the `<base_link>` and `<link>` tags, which replace the `<dynamic>` tag. A definition of a robot has to contain at least one link, called the base link. Only one base link can exist in a kinematic tree.

3.9.2 Joints

Joints are connections between links that define the relative position and orientation of links, as well as the allowed directions of motion (degrees of freedom). The `<joint>` tag is used to define joints in the XML syntax. There are three types of joints implemented in the *Stonefish* library:

1. Fixed `type="fixed"` - all degrees of freedom are locked
2. Prismatic `type="prismatic"` - one linear degree of freedom
3. Revolute `type="revolute"` - one angular degree of freedom

3.9.3 Defining a robot

The definition of the robot comprises multiple links, joints and attached devices. If the robot is defined using the XML syntax, the order of the definitions of these components does not matter. However, using the C++ code, there are few steps which have to be completed in the following order:

1. Defining links
2. Defining joints (building the structure)
3. Creating actuators, sensors and communication devices

4. Attaching actuators, sensors and communication devices
5. Adding robot to the simulation scenario.

Below, an example of defining a complete robot is presented, first using the XML syntax and later using its C++ twin. It is assumed that the material "Steel" and the look "Green" were defined before.

```xml
<robot name="Robot" fixed="false" self_collisions="false">
    <base_link name="Base" type="sphere" physics="surface">
        <dimensions radius="0.2"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <material name="Steel"/>
        <look name="Green"/>
    </base_link>
    <link name="Link1" type="box" physics="surface">
        <dimensions xyz="0.1 0.02 0.5"/>
        <origin xyz="0.0 0.0 -0.2" rpy="0.0 0.0 3.14159"/>
        <material name="Steel"/>
        <look name="Green"/>
    </link>
    <link name="Link2" type="box" physics="surface">
        <dimensions xyz="0.1 0.02 0.5"/>
        <origin xyz="0.0 0.0 -0.2" rpy="0.0 0.0 3.14159"/>
        <material name="Steel"/>
        <look name="Green"/>
    </link>
    <joint name="Joint1" type="revolute">
        <parent name="Base"/>
        <child name="Link1"/>
        <origin xyz="0.0 0.25 -0.2" rpy="0.0 0.0 0.0"/>
        <axis xyz="0.0 1.0 0.0"/>
    </joint>
    <joint name="Joint2" type="revolute">
        <parent name="Base"/>
        <child name="Link2"/>
        <origin xyz="0.0 -0.25 -0.2" rpy="0.0 0.0 0.0"/>
        <axis xyz="0.0 1.0 0.0"/>
    </joint>
    <actuator name="Servo" type="servo">
        <controller position_gain="1.0" velocity_gain="1.0" max_torque="10.0"/>
        <joint name="Joint1"/>
    </actuator>
    <sensor name="Encoder" type="encoder">
        <history samples="1000"/>
        <joint name="Joint2"/>
    </sensor>
    <sensor name="IMU" type="imu" rate="10.0">
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <link name="Link2"/>
    </sensor>
    <world_transform xyz="0.0 0.0 -2.0" rpy="0.0 0.0 0.0"/>
</robot>
```

```
//1. Defining links
sf::Sphere* base = new sf::Sphere("Base", 0.2, sf::I4(), "Steel",␣
↪sf::BodyPhysicsType::SURFACE, "Green");
sf::Box* link1 = new sf::Box("Link1", sf::Vector3(0.1, 0.02, 0.5),␣
↪sf::Transform(sf::Quaternion(M_PI_2, 0.0, 0.0), sf::Vector3(0.0, 0.0, -0.2)), "Steel
↪", sf::BodyPhysicsType::SURFACE, "Green");
```

(continues on next page)

(continued from previous page)

```
sf::Box* link2 = new sf::Box("Link2", sf::Vector3(0.1, 0.02, 0.5),␣
↪sf::Transform(sf::Quaternion(M_PI_2, 0.0, 0.0), sf::Vector3(0.0, 0.0, -0.2)), "Steel
↪", sf::BodyPhysicsType::SURFACE, "Green");

std::vector<sf::SolidEntity*> links;
links.push_back(link1);
links.push_back(link2);

//2. Building the structure
sf::Robot* robot = new sf::Robot("Robot", false);
robot->DefineLinks(base, links);
robot->DefineRevoluteJoint("Joint1", "Base", "Link1", sf::Transform(sf::IQ(),␣
↪sf::Vector3(0.0, 0.25, -0.2)), sf::Vector3(0.0, 1.0, 0.0), std::make_pair(1.0, -1.
↪0));
robot->DefineRevoluteJoint("Joint2", "Base", "Link2", sf::Transform(sf::IQ(),␣
↪sf::Vector3(0.0, -0.25, -0.2)), sf::Vector3(0.0, 1.0, 0.0), std::make_pair(1.0, -1.
↪0));

//3. Creating actuators and sensors
sf::ServoMotor* srv = new sf::ServoMotor("Servo", 1.0, 1.0, 10.0);
sf::IMU* imu = new sf::IMU("IMU", 10.0);
sf::RotaryEncoder* enc = new sf::RotaryEncoder("Encoder", -1.0, 1000);

//4. Attaching actuators and sensors
robot->AddJointActuator(srv, "Joint1");
robot->AddJointSensor(enc, "Joint2");
robot->AddLinkSensor(imu, "Link2", sf::I4());

//5. Adding robot to the simulation scenario
AddRobot(robot, sf::Transform(sf::IQ(), sf::Vector3(0.0, 0.0, -2.0)));
```

3.10 Sensors

A rich set of sensor simulations is available in the *Stonefish* library, including the ones specific for the marine robotics. The implemented sensors can be divided into three groups: the joint sensors, the link sensors and the vision sensors. Each sensor type is described below to understand its operation and the way to include it in the simulation scenario. Most of the sensors include appropriate noise models which can be optionally enabled.

Warning: Sensors can only be created in connection with a definition of a *robot*, because they have to be attached to a robot's joint or link.

Note: In the following sections, description of each specific sensor implementation is accompanied with an example of sensor instantiation through the XML syntax and the C++ code. It is assumed that the XML snippets are located inside the definition of a robot. In case of C++ code, it is assumed that an object `sf::Robot* robot = new sf::Robot(...);` was created before the sensor definition.

3.10.1 Joint sensors

The joint sensors are attached to the robot's joints and measure their internal states. All of them share the following properties:

1) **Name:** unique string
2) **Rate:** sensor update frequency [Hz] (optional)
3) **Type:** type of the sensor
4) **History length**: the size of the measurement buffer
5) **Joint name**: the name of the robot joint that the sensor is attached to

```
<sensor name="{1}" rate="{2}" type="{3}">
   <!-- specific definitions here -->
   <history samples="{4}"/>
   <joint name="{5}"/>
</sensor>
```

Rotary encoder

The rotary encoder measures the rotation angle of a specified joint. It does not have any specific properties.

```
<sensor name="Encoder" rate="10.0" type="encoder">
    <history samples="100"/>
    <joint name="Joint1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/RotaryEncoder.h>
sf::RotaryEncoder* encoder = new sf::RotaryEncoder("Encoder", 10.0, 100);
robot->AddJointSensor(encoder, "Joint1");
```

Torque (1-axis)

The torque sensor measures the torque excerted on a specified joint. The measurement range and the standard deviation of the measured torque can be optionally defined.

```
<sensor name="Torque" rate="100.0" type="torque">
    <range torque="10.0"/>
    <noise torque="0.05"/>
    <history samples="100"/>
    <joint name="Joint1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Torque.h>
sf::Torque* torque = new sf::Torque("Torque", 100.0, 100);
torque->setRange(10.0);
torque->setNoise(0.05);
robot->AddJointSensor(torque, "Joint1");
```

Force-torque (6-axis)

The force-torque sensor is a 6-axis sensor located in a specified joint. It measures force and torque in all three directions of a Cartesian reference frame, attached to the child link of the joint. The measurement range for each of the sensor channels and the standard deviation of measurements can be optionally defined.

```
<sensor name="FT" rate="100.0" type="forcetorque">
    <range force="10.0 10.0 100.0" torque="1.0 1.0 2.0"/>
    <noise force="0.5" torque="0.05"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <history samples="1"/>
    <joint name="Joint1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/ForceTorque.h>
sf::ForceTorque* ft = new sf::ForceTorque("FT", sf::I4(), 100.0, 1);
ft->setRange(sf::Vector3(10.0, 10.0, 100.0), sf::Vector3(1.0, 1.0, 2.0));
ft->setNoise(0.5, 0.05);
robot->AddJointSensor(ft, "Joint1");
```

3.10.2 Link sensors

The link sensors measure motion related or environment related quantities. They are attached to the robot's links. All of them share the following properties:

1) **Name:** unique string
2) **Rate:** sensor update frequency [Hz] (optional)
3) **Type:** type of the sensor
4) **History length**: the size of the measurement buffer
5) **Origin:** the transformation from the link frame to the sensor frame
6) **Link name**: the name of the robot link that the sensor is attached to

```
<sensor name="{1}" rate="{2}" type="{3}">
   <!-- specific definitions here -->
   <history samples="{4}"/>
   <origin xyz="{5a}" rpy="{5b}"/>
   <link name="{6}"/>
</sensor>
```

IMU

The inertial measurement unit (IMU) measures the orientation and the angular velocities of the link. The angular velocity measurement range and the standard deviation of angle and angular velocity measurements can be optionally defined.

```
<sensor name="IMU" rate="10.0" type="imu">
    <range angular_velocity="0.5"/>
    <noise angle="0.1" angular_velocity="0.05"/>
    <history samples="1"/>
    <origin xyz="0.1 0.0 0.0" rpy="0.0 0.0 0.0"/>
```

(continues on next page)

(continued from previous page)

```
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/IMU.h>
sf::IMU* imu = new sf::IMU("IMU", 10.0, 1);
imu->setRange(0.5);
imu->setNoise(0.1, 0.05);
robot->AddLinkSensor(imu, "Link1", sf::Transform(sf::Quaternion(0.0, 0.0, 0.0),␣
↪sf::Vector3(0.1, 0.0, 0.0));
```

Odometry

The odometry sensor is a virtual sensor which can be used to obtain the navigation ground truth or emulate navigation system with errors. It measures position, linear velocities, orientation and angular velocities. Standard deviation for each of the quantities can be optionally specified.

```
<sensor name="Odometry" rate="10.0" type="odometry">
    <noise position="0.05" velocity="0.01" angle="0.1" angular_velocity="0.05"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Odometry.h>
sf::Odometry* odom = new sf::Odometry("Odometry", 10.0, 1);
odom->setNoise(0.05, 0.01, 0.1, 0.05);
robot->AddLinkSensor(odom, "Link1", sf::I4());
```

GPS

The global positioning system (GPS) sensor measures the position of the link in the world frame and converts it into the geographic coordinates and altitude. This sensor works only when above the water level. Optionally, it is possible to define the standard deviation of the position error in the NED frame.

```
<sensor name="GPS" rate="1.0" type="gps">
    <noise ned_position="0.5"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/GPS.h>
sf::GPS* gps = new sf::GPS("GPS", 1.0, 1);
gps->setNoise(0.5);
robot->AddLinkSensor(gps, "Link1", sf::I4());
```

Compass

The compass is measuring the heading of the robot. Optionally it is possible to define standard deviation of the measurement.

```
<sensor name="Compass" rate="1.0" type="compass">
    <noise heading="0.1"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Compass.h>
sf::Compass* comp = new sf::Compass("Compass", 1.0, 1);
comp->setNoise(0.1);
robot->AddLinkSensor(comp, "Link1", sf::I4());
```

Pressure

The pressure sensor measures the gauge pressure underwater. Pressure range as well as standard deviation of the measurement can be defined.

```
<sensor name="Pressure" rate="1.0" type="pressure">
    <range pressure="10000.0"/>
    <noise pressure="0.1"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Pressure.h>
sf::Pressure* press = new sf::Pressure("Pressure", 1.0, 1);
press->setRange(10000.0);
press->setNoise(0.1);
robot->AddLinkSensor(press, "Link1", sf::I4());
```

Doppler velocity log (DVL)

The Doppler velocity log (DVL) is a classic marine craft sensor, used for measuring vehicle velocity as well as water velocity. The current implementation of DVL in the Stonefish library is using four acoustic beams to determine the altitude above terrain. The shortest distance is reported. Moreover, it provides robot velocity along all three Cartesian axes. The velocity is calculated based on the simulation of motion rather than the Doppler effect, which may be improved in future. It is possible to specify sensor operating range in terms of the altitude limits as well as the maximum measured velocity. Noise can be added to the measurements as well.

```
<sensor name="DVL" rate="10.0" type="dvl">
    <specs beam_angle="30.0"/>
    <range velocity="10.0 10.0 5.0" altitude_min="0.5" altitude_max="50.0"/>
    <noise velocity="0.1" altitude="0.03"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/DVL.h>
sf::DVL* dvl = new sf::DVL("DVL", 30.0, 10.0, 1);
dvl->setRange(sf::Vector3(10.0, 10.0, 5.0), 0.5, 50.0);
```

(continues on next page)

(continued from previous page)

```
dvl->setNoise(0.1, 0.03);
robot->AddLinkSensor(dvl, "Link1", sf::I4());
```

Profiler

The profiler is a simple acoustic or laser-based device that measures distance to the obstacles by shooting a narrow beam, rotating around an axis, in one plane. Each measurement is a single distance, followed by a change in beam rotation. The specification of the profiler device requires two parameters: the field of view (FOV) and the number of rotation steps. It is also possible to define measured distance limits and measurement noise.

```
<sensor name="Profiler" rate="10.0" type="profiler">
    <specs fov="120.0" steps="128"/>
    <range distance_min="0.5" distance_max="10.0"/>
    <noise distance="0.05"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Profiler.h>
sf::Profiler* prof = new sf::Profiler("Profiler", 120.0, 128, 10.0, 1);
prof->setRange(0.5, 10.0);
prof->setNoise(0.05);
robot->AddLinkSensor(prof, "Link1", sf::I4());
```

Multi-beam sonar

The multi-beam sonar is an acoustic device that measures distance to obstacles by sending acoustic pulses. It utilises multiple acoustic beams, arranged in a planar fan shape. This implementation neglects the beam parameters and resorts to tracing a single ray per beam. More advanced sonar simulations can be found under vision sensors. The output of the multibeam is a planar distance map, in a cylindrical coordinate system. The specification of the multibeam device requires two parameters: the field of view (FOV) and the number of angle steps (beams). It is also possible to define measured distance limits and measurement noise.

```
<sensor name="Multibeam" rate="1.0" type="multibeam1d">
    <specs fov="120.0" steps="128"/>
    <range distance_min="0.5" distance_max="50.0"/>
    <noise distance="0.1"/>
    <history samples="1"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/scalar/Multibeam.h>
sf::Multibeam* mb = new sf::Multibeam("Multibeam", 120.0, 128, 1.0, 1);
mb->setRange(0.5, 50.0);
mb->setNoise(0.1);
robot->AddLinkSensor(mb, "Link1", sf::I4());
```

3.10.3 Vision sensors

The simulation of vision sensors is based on images generated by the GPU. In case of a typical color camera it means rendering the scene as usual and downloading the frame from the GPU. In case of a more sophisticated sensor like a forward-looking sonar (FLS) it means generating a special input image from the scene data, processing this image to account for the properties of the sensor, and generating an output display image. All processing is fully GPU-based for the ultimate performance. The vision sensors are attached to the robot's links. All of them share the following properties:

1) **Name:** unique string
2) **Rate:** sensor update frequency [Hz] (optional)
3) **Type:** type of the sensor
4) **Origin:** the transformation from the link frame to the sensor frame
5) **Link name**: the name of the robot link that the sensor is attached to

```
<sensor name="{1}" rate="{2}" type="{3}">
   <!-- specific definitions here -->
   <origin xyz="{4a}" rpy="{4b}"/>
   <link name="{5}"/>
</sensor>
```

Note: Sensor update frequency (rate) is not used in sonar simulations. The actual rate is determined by the maximum sonar range and the sound velocity in water.

Color camera

The color camera is a virtual pinhole camera. The output image is rendered using the standard mode, the same as the visualisation in the main window.

```
<sensor name="Cam" rate="10.0" type="camera">
    <specs resolution_x="800" resolution_y="600" horizontal_fov="60.0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/vision/ColorCamera.h>
sf::ColorCamera* cam = new sf::ColorCamera("Cam", 800, 600, 60.0, 10.0);
robot->AddVisionSensor(cam, "Link1", sf::I4());
```

Depth camera

The depth camera captures a linear depth image. The output image is a grayscale floating-point bitmap, where black and white colors representing the minimum and maximum captured depth respectively.

```
<sensor name="Dcam" rate="5.0" type="depthcamera">
    <specs resolution_x="800" resolution_y="600" horizontal_fov="60.0" depth_min="0.2
↪" depth_max="10.0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/vision/DepthCamera.h>
sf::DepthCamera* cam = new sf::DepthCamera("Dcam", 800, 600, 60.0, 0.2, 10.0, 5.0);
robot->AddVisionSensor(cam, "Link1", sf::I4());
```

Forward-looking sonar (FLS)

The forward-looking sonar (FLS) is an acoustic device utilising multiple acoustic beams arranged in a planar fan pattern, to generate an acoustic echo intensity map in cylindrical coordinates. This map can be used to detect obstacles or map underwater structures. A characteristic property of this kind of sonar is that the beam width perpendicular to the fan plane is significant, leading to multiple echoes from different beam parts which get projected on the same line.

```
<sensor name="FLS" type="fls">
    <specs beams="512" bins="500" horizontal_fov="120.0" vertical_fov="30.0"/>
    <settings range_min="0.5" range_max="10.0" gain="1.1"/>
    <display colormap="hot"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/vision/FLS.h>
sf::FLS* fls = new sf::FLS("FLS", 512, 500, 120.0, 30.0, 0.5, 10.0,␣
↪sf::ColorMap::HOT);
fls->setGain(1.1);
robot->AddVisionSensor(fls, "Link1", sf::I4());
```

Mechanical scanning imaging sonar (MSIS)

The mechanical scanning imaging sonar (MSIS) is an acoustic device utilising a single rotating acoustic beam. The beam rotates in one plane and generates an acoustic echo intensity map in cylindrical coordinates. This map can be used to detect obstacles or map underwater structures. This kind of sonar produces images similar to the FLS, but due to the rotation of the beam the image is corrupted by the robot's motion.

```
<sensor name="MSIS" type="msis">
    <specs step="0.25" bins="500" horizontal_beam_width="2.0" vertical_beam_width="30.
↪0"/>
    <settings range_min="0.5" range_max="10.0" rotation_min="-50.0" rotation_max="50.0
↪" gain="1.5"/>
    <display colormap="hot"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/vision/MSIS.h>
sf::MSIS* msis = new sf::MSIS("MSIS", 0.25, 500, 2.0, 30.0, -50.0, 50.0, 0.5, 10.0,␣
↪sf::ColorMap::HOT);
msis->setGain(1.5);
robot->AddVisionSensor(msis, "Link1", sf::I4());
```

Side-scan sonar (SSS)

The side-scan sonar (SSS) is an acoutic device with two tranducers, located symmetrically on the robot's hull, with a specified angular separation. The transducers are commonly pointing to the seafloor and allow for fast and detailed

mapping of large areas. Each of the transducers emits and receives one beam, creating one line of an acoustic image. The display of the acoustic map is done by adding subsequent lines in a "waterfall" fashion.

```
<sensor name="SSS" type="sss">
    <specs bins="500" lines="400" horizontal_beam_width="2.0" vertical_beam_width="50.
↪0" vertical_tilt="60.0"/>
    <settings range_min="1.0" range_max="100.0" gain="1.2"/>
    <display colormap="hot"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</sensor>
```

```
#include <Stonefish/sensors/vision/SSS.h>
sf::SSS* sss = new sf::SSS("SSS", 500, 400, 50.0, 2.0, 60.0, 1.0, 100.0,␣
↪sf::ColorMap::HOT);
sss->setGain(1.2);
robot->AddVisionSensor(sss, "Link1", sf::I4());
```

3.11 Actuators

A set of common actuator models is available in the *Stonefish* library, including the ones specific for the marine robotics. The implemented actuators can be divided into two groups: the joint actuators and the link actuators. Each actuator type is described below to understand its operation and the way to include it in the simulation scenario.

Warning: Actuators can only be created in connection with a definition of a *robot*, because they have to be attached to a robot's joint or link.

Note: In the following sections, description of each specific actuator implementation is accompanied with an example of actuator instantiation through the XML syntax and the C++ code. It is assumed that the XML snippets are located inside the definition of a robot. In case of C++ code, it is assumed that an object `sf::Robot* robot = new sf::Robot(...);` was created before the actuator definition.

3.11.1 Joint actuators

The joint actuators are attached to the robot's joints and they apply forces or torques between the links. They share a set of common properties:

1) **Name:** unique string
2) **Type:** type of the actuator
3) **Joint name**: the name of the robot joint that the actuator is attached to

```
<actuator name="{1}" type="{2}">
   <!-- specific definitions here -->
   <joint name="{3}"/>
</sensor>
```

Servomotor

A servomotor is an electric motor connected with control and power circuits that allow for controlling it in different modes: torque, position or velocity.

```
<actuator name="Servo" type="servo">
    <controller position_gain="1.0" velocity_gain="0.5" max_torque="10.0"/>
    <joint name="Joint1"/>
</actuator>
```

```
#include <Stonefish/actuators/Servo.h>
sf::Servo* srv = new sf::Servo("Servo", 1.0, 0.5, 10.0);
robot->AddJointActuator(srv, "Joint1");
```

3.11.2 Link actuators

The link actuators are attached to the robot's links and they apply forces or torques to the links. They share a set of common properties:

1) **Name:** unique string
2) **Type:** type of the actuator
3) **Origin:** the transformation from the link frame to the actuator frame
4) **Link name**: the name of the robot link that the actuator is attached to

```
<actuator name="{1}" type="{2}">
   <!-- specific definitions here -->
   <origin xyz="{3a}" rpy="{3b}"/>
   <link name="{4}"/>
</sensor>
```

Propeller

A propeller is an actuator working in atmosphere, representing an airplane propeller driven by a motor.

```
<actuator name="Prop" type="propeller">
    <specs thrust_coeff="0.45" torque_coeff="0.02" max_rpm="1000" inverted="false"/>
    <propeller diameter="0.5" right="true">
        <mesh filename="propeller.obj" scale="1.0"/>
        <material name="Steel"/>
        <look name="Red"/>
    </propeller>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</actuator>
```

```
#include <Stonefish/actuators/Propeller.h>
sf::Polyhedron* propMesh = new sf::Polyhedron("PropMesh", sf::GetDataPath() +
↪"propeller.obj", 1.0, sf::I4(), "Steel", sf::BodyPhysicsType::AERODYNAMIC, "Red");
sf::Propeller* prop = new sf::Propeller("Prop", propMesh, 0.5, 0.45, 0.02, 1000, true,
↪ false);
robot->AddLinkActuator(prop, "Link1", sf::I4());
```

Thruster

A thruster is an actuator working underwater, representing an underwater thruster with a propeller.

```
<actuator name="Thruster" type="thruster">
    <specs thrust_coeff="0.45" thrust_coeff_backward="0.35" torque_coeff="0.02" max_
↪rpm="1000" inverted="false"/>
    <propeller diameter="0.2" right="true">
        <mesh filename="propeller.obj" scale="1.0"/>
        <material name="Steel"/>
        <look name="Red"/>
    </propeller>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</actuator>
```

```
#include <Stonefish/actuators/Thruster.h>
sf::Polyhedron* propMesh = new sf::Polyhedron("PropMesh", sf::GetDataPath() +
↪"propeller.obj", 1.0, sf::I4(), "Steel", sf::BodyPhysicsType::SUBMERGED, "Red");
sf::Thruster* th = new sf::Thruster("Thruster", propMesh, 0.2, std::make_pair(0.45, 0.
↪35), 0.02, 1000, true, false);
robot->AddLinkActuator(th, "Link1", sf::I4());
```

Variable buoyancy system (VBS)

A variable buoyancy system (VBS) is a container with an elastic wall, which can be filled with gas under pressure to change its volume and thus its buoyancy. It is used to control the depth of the robot. The VBS is defined by providing a set of meshes representing its states between minimum and maximum volume. Between these shapes the volume is interpolated linearly. In the current implementation, due to the limitations of the physics engine, the inertia of the water filling the container is not taken into account when computing dynamic forces.

```
<actuator name="VBS" type="vbs">
    <volume initial="0.5">
        <mesh filename="empty.obj"/>
        <mesh filename="half.obj"/>
        <mesh filename="full.obj"/>
    </volume>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</actuator>
```

```
#include <Stonefish/actuators/VariableBuoyancy.h>
std::vector<std::string> meshes;
meshes.push_back(sf::GetDataPath() + "empty.obj");
meshes.push_back(sf::GetDataPath() + "half.obj");
meshes.push_back(sf::GetDataPath() + "full.obj");
sf::VariableBuoyancy* vbs = new sf::VariableBuoyancy("VBS", meshes, 0.5);
robot->AddLinkActuator(vbs, "Link1", sf::I4());
```

Light

A light is a special actuator that does not generate any forces but represents an omnidirectional or spot light. It can be used to simulate artificial lighting in dark environments, light beacons etc.

```
<actuator name="Light" type="light">
    <specs radius="0.1" cone_angle="30.0" illuminance="1000.0"/>
    <color temperature="5600.0"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</actuator>
```

```
#include <Stonefish/actuators/Light.h>
sf::Light* lt = new sf::Light("Light", 0.1, 30.0, sf::Color::BlackBody(5600.0), 1000.
↪0);
robot->AddLinkActuator(lt, "Link1", sf::I4());
```

3.12 Communication devices

Communication devices were included in the *Stonefish* library to account for the delays and directional character of different communication interfaces, in different mediums. They can be attached to the links of a robot, to a dynamic or static body, or to the world origin.

The common properties of communication devices are:

1) **Name**: unique string
2) **Device ID**: unique number identifying the device
3) **Type**: type of the communication device
4) **Origin**: position and orientation of the device frame with respect to the parent frame
5) **Name of link**: name of the link of the robot the device is attached to (optional)
6) **Name of body**: name of the body the device is attached to (static or dynamic; optional)

```
<comm name="{1}" device_id="{2}" type="{3}">
    <!-- specific definitions here -->
    <origin xyz="{4a}" rpy="{4b}"/>
    <link name="{5}"/>
    <body name="{6}"/>
</comm>
```

Warning: Only one of the options: 5 or 6, can be used. If neither `link` nor `body` tag is specified, the communication device is considered to be attached to the world origin.

Note: In the following sections, description of each specific implementation of a communication device is accompanied with an example of its instantiation through the XML syntax and the C++ code. It is assumed that the XML snippets are located inside the definition of a robot. In case of C++ code, it is assumed that an object `sf::Robot* robot = new sf::Robot(...);` was created before the device definition.

3.12.1 Acoustic modem

An acoustic modem is an underwater communication device based on an acoustic transducer.

```
<comm name="Modem" device_id="5" type="acoustic_modem">
    <specs horizontal_fov="360.0" vertical_fov="120.0" range="1000.0"/>
    <connect device_id="9"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</comm>
```

```
#include <Stonefish/comms/AcousticModem.h>
sf::AcousticModem* modem = new sf::AcousticModem("Modem", 5, 360.0, 120.0, 1000.0);
modem->Connect(9);
robot->AddComm(modem, "Link1", sf::I4());
```

3.12.2 USBL

The ultra short baseline (USBL) is a device based on a tightly packed array of underwater acoustic transducers. It can be used for underwater communication as well as for localization of the signal source in 3D space. User can optionally define the standard deviation of the angle and range measurments, similarily to the sensor definitions. Another feature of the USBL implementation is an automatic ping function used to update the measurements at a specified rate.

```
<comm name="USBL" device_id="5" type="usbl">
    <specs horizontal_fov="360.0" vertical_fov="160.0" range="1000.0"/>
    <connect device_id="9"/>
    <autoping rate="1.0"/>
    <noise range="0.05" angle="0.02"/>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    <link name="Link1"/>
</comm>
```

```
#include <Stonefish/comms/USBL.h>
sf::USBL* usbl = new sf::USBL("USBL", 5, 360.0, 160.0, 1000.0);
usbl->Connect(9);
usbl->EnableAutoPing(1.0);
usbl->setNoise(0.05, 0.02, 0.0, 0.0);
robot->AddComm(usbl, "Link1", sf::I4());
```

3.13 Special functionality

A list of special functions of the *Stonefish* library is gathered below. It includes tools, which can be useful in the analysis of the behaviour of simulated systems, not representing real world devices.

3.13.1 Contacts

The contact recording function enables capturing the history of contact points between two selected bodies (dynamic or static). The recorded contact points can optionally be displayed in the visualisation window, using one of the selected representations: path, slip (velocity) or force. Moreover, the visualisation is defined separately for each of the bodies.

An example of using a contact recording feature is presented below:

```
<contact name="Contact1">
    <bodyA name="Body1" display="path"/>
    <bodyB name="Body2"/>
    <history points="1000"/>
</contact>
```

```
#include <Stonefish/sensors/Contact.h>
sf::SolidEntity* body1 = ...;
sf::StaticEntity* body2 = ...;
sf::Contact* cnt = new sf::Contact("Contact1", body1, body2, 1000);
cnt->setDisplayMask(CONTACT_DISPLAY_PATH_A);
AddContact(cnt);
```

3.13.2 Soft collision

Collisions of all bodies defined in the simulation scenario are considered rigid by default. Sometimes it is important to improve the stability of the collision response, e.g., in case of high impacts, or introduce softness, which can simulate linearly deformable materials. This can be achieved by extending the definition of a selected dynamic body as follows:

```
<dynamic>
    <!-- all standard definitions -->
    <contact stiffness="1000.0" damping="0.5"/>
</dynamic>
```

```
sf::SolidEntity* solid = ...;
solid->SetContactProperties(true, 1000.0, 0.5);
```

3.14 Changelog

3.14.1 1.1

- Removed external dependence on the Bullet Physics Library and included necessary parts in the source tree
- Updated the mathematical models of the thruster and the propeller actuators
- Optimised computation of the geometry-based hydrodynamics/aerodynamics
- Implemented new visualisation of underwater currents (water velocity field)
- Fixed crashes when trying to create marine actuators in a simulation without ocean

3.14.2 1.0

- Fully GPU-based simulation of mechanical scanning imaging sonar (MSIS)
- Improvements in all sonar simulations
- Significant improvement to DVL performance when heightfield terrain is used
- Heightfield terrain now supports 16 bit heightmaps
- New syntax for loading ocean and atmosphere definitons using the XML parser
- Support for arguments passed to the included files

- New, complete, beautiful documentation generated with Sphinx

3.14.3 0.9

- Moved to the OpenGL 4.3 functionality (compute shaders)
- Complete rewrite of the ocean/underwater rendering pipeline
- Light absorption and scattering in water based on Jerlov measurements
- Full support of photo-reallistic sky and sunlight as well as point and spot lights
- New, linear tree based, automatic LOD algorithm
- New automatic exposure (histogram based) and anti-aliasing (FXAA) algorithms
- Logarythmic depth buffer for planet scale rendering without precision issues
- Fully GPU-based simulation of forward-looking sonar (FLS)
- Fully GPU-based simulation of side-scan sonnar (SSS)
- Normal mapping to enable high resolution surface details
- Faster download of data from the GPU memory
- Scheduling of the rendering of multiple views
- Reallistic measurement of the drawing time
- Interactive selection outline in 3D view
- OpenGL function handlers provided through GLAD (dropped outdated GLEW)
- General cleaning of code and refactoring
- Dozens of bug fixes

3.14.4 Origins

This project started when I was writing my PhD thesis and needed a realtime simulator for a balancing mono-wheel robot. The simulator not only had to be fast but also deliver high fidelity results. After investigating commercial solutions I have reached the conculsion that I need to implement my own tool becasue simulation times were prohibitively long and no direct interaction with the robot was possible. I decided to use Bullet Physics library and build a simulator capable of computing multi-body dynamics with an analytic tyre-ground collision model, in realitime. Thanks to this simulator I was able to implement my whole control system in a virtual environment and simulate the robot in an interactive way, which allowed me to finish my PhD thesis.

During my PhD studies I had a brief adventure with underwater robotics and after I finished my PhD I started working in this field. Being mostly interested in control design, I have realised that a modern simulator for underwater robots is missing. That is how I started exteding *Stonefish* with marine robotics features and regularily using it in my research. I saw that this work can be of benefit for the whole marine robotics community and decided to release it as open-source software.

3.15 License

Stonefish is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/.