# Digital Twin Platform for Unmanned Surface Vehicles

Simulating Unmanned Surface Vehicles for robotics and control developers

**Authors**
Edvart Grüner Bjerke
Isak Lars Haukeland

**Abstract**
Unmanned Surface Vehicles (USVs) are an emerging technology at the intersection of robotics, machine learning, Guidance, Navigation and Control (GNC) and maritime technology. Simulators and digital twin solutions can be massively helpful in enabling rapid development at reduced costs, as they allow for prototyping and validation without the need of a fully functioning physical vessel. In this thesis, a digital twin & simulation platform is proposed. The simulation environment features fluid-to-solid interactions, supports a variety of sensors, and fully integrates with the Robot Operating System (ROS). The platform is implemented in Unity, with high-fidelity graphics, and can be used to generate synthetic data for validation and training of machine learning models. The simulator is easily customizable, allowing for quick setup of virtual scenarios (e.g. docking, collision avoidance), facilitating the development of autonomous algorithms. The simulator has been successfully tested on common robotics operations such as Simultaneous Localization and Mapping and autonomous navigation, demonstrating its efficacy and reliability as a valuable tool for robotics development.

A thesis presented for the degree of
Bachelor of Data Science and Electrical Engineering

Faculty of Technology, Natural Sciences and Maritime Sciences
University of South-Eastern Norway
Horten, Norway
May 22, 2024

# Acknowledgments

We, the authors, are grateful for the contributions and help of our counselors. Main counselor Karl Thomas Hjelmervik of the University of South-Eastern Norway scientific faculty has delivered instrumental advice and structure through our weekly group meetings. His feedback on our work has been especially helpful towards the final deadline, focusing our message and balancing the sections.

Also from the University of South-Eastern Norway, Karina Bakkeløkken Hjelmervik has recommended and lent us relevant books to reference in our background theory section.

We wish to thank our external counselor Torbjørn Houge of Maritime Robotics AS. He has supplied us with insight into the current bleeding edge of autonomous surface vessel industry. This helped us in evaluating the potential of our work, along with future developments that would be beneficial in an industry setting.

# Table of Contents

# 1. Introduction

Developments in machine learning and artificial intelligence enable autonomous vehicles through technologies like computer vision [1] and reinforcement learning. This development has enabled a new class of maritime vehicles, with products like Kongsberg's Autonomous Underwater Vehicle *HUGIN* [2] or the ASKO barges for transporting groceries made in collaboration with Kongsberg Maritime [3]. Another actor in the autonomous surface vessel industry is Maritime Robotics [4], with their Otter [5] and Mariner [6] boats, which carry out marine surveying and other oceanographic tasks. Other applications include defence operations [7] and search & rescue missions [8, 9].

As the field of robotics moves away from repetitive tasks in static environments and into more complex operations in dynamic spaces where safety and autonomy are crucial, simulators and digital twins become increasingly useful to the developer [10]. USVs exemplify this shift, as they must adapt to changing environments, cooperate with other autonomous or human-operated vehicles and comply with maritime regulations (e.g. COLREGS [11]).

The development of USVs presents some key challenges that highlight the need for rich simulation environments. Economically, simulators offer a cost-effective way of testing and iterating on the software design of the vehicle, as they mitigate the need for physical prototypes and field tests, which are typically expensive. Simulators are also highly practical, as they allow developers to create and test a wide range of scenarios that would be difficult, dangerous, or impractical to recreate in real life. They also allow for concurrency in the development process, as multiple teams can work on the virtual models simultaneously, testing different aspects of the USV's performance and software, in different scenes and scenarios.

The goal of this thesis is to propose a user-friendly simulation platform that can be used to develop and test robotics algorithms, generate synthetic data and simulate a variety of sensors. The environment should be easily customizable and support high quality rendering. It should also include visually satisfying water physics for floating objects, with physically-based buoyancy and hydrodynamics.

The digital twin platform is implemented in the game engine Unity [12], which provides a user-friendly suite of tools for the creation of custom scenes and interactions. To visualize the world and provide the basis for our hydrodynamic system, we leverage the High Definition Rendering Pipeline [13] (HDRP) and its water system [14], capable of large scale, real-time ocean simulation. The water system provides a rich interface of parameters that can be tuned to obtain the desired ocean behaviour.

The details of the water interaction implementation are laid out in Section 4.2.1. In summary, we provide a configurable physics system for water interaction, including buoyancy and hydrodynamic forces. The buoyancy system may be configured to use surface elements (surfels) or volume elements (voxels), which calculate buoyancy forces on floating objects by sampling the height of the neighbouring wave field. Hydrodynamic forces are computed by iterating over surfels or by a state-space approximation, and the pros and cons of these implementations are discussed.

In addition to the visual environment and physical interaction, the simulation of sensors is essential for a useful digital twin, as most software for autonomy relies on input data gathered from physical sensors. The sensors included in our platform and

their respective implementations are detailed in Section 4.4. We also implement a basic propulsion system, which allows the user to specify the desired thruster configuration, and provides basic parameters for tuning of thruster dynamics.

An important feature for the user of a simulator is the ability to extract data from it at runtime. For this prupose, we leverage the ROS-TCP-Connector [15], which provides an API for integrating the Unity environment with the Robot Operating System (ROS) [16], a framework for developing robotic systems. This allows developers to read sensor data and send actuator commands which affect the vessel in real-time.

To illustrate the effectiveness of our platform, we present a few common applications of the simulator, mirroring those typically found on real unmanned surface vehicles:

- Simultaneous Localization and Mapping (SLAM)
- Object detection
- Synthetic data generation
- Autonomous Navigation

The digital twin & simulation platform is released as free and open source software under the MIT license. The source code can be found at: `https://github.com/edvart-ros/unity_asv_sim`

## 2. Literature Review

This section aims to collect and present the relevant literature from the fields of computer physics simulation, real and approximated water physical behaviour, the implementation of the Unity water system, and various methods for sampling the water surface and translating this into rotation/translation parameters for our ship, applied either at the center of gravity or to each of a sub-component of the mesh (triangles, boxes).

The paper "A survey of ocean simulation and rendering techniques in computer graphics" by Darles et al. [17] is more than ten years old, but it presents advanced techniques still significant today. It refers to several other papers and describe their discoveries and how they relate to the bigger picture of ocean simulation, and serves as good background knowledge for understanding the approach taken by Unity for their water system.

In "Realistic Buoyancy Model for Real-Time Applications" by J. M. Bajo, G. Patow and C. A. Delrieux [18], the authors provide a method for simulating buoyancy on the Graphics Processing Unit (GPU), and incorporate the complexity of simulating a sinking object by adding a weight force to different parts of a 3D model. We incorporate some of their approach into one of our buoyancy models, and replicate their validation experiment (Section 5.1.), where simulation results are compared to an analytical solution. This paper also demonstrates how the CPU-based buoyancy system presented in this thesis may be improved in performance and scalability by leveraging GPU parallelization.

The new Unity HDRP water system is documented by authors Adrien de Tocqueville et al. in their article on the Unity blogs website [14]. There, they describe the capabilities of the system, and how to get started using it. They go on to explain their use of the Fast-Fourier Transform to generate the waves procedurally, and divides the waves into three bands for granular control of the different frequencies. The computational demand is a rendering time of 4 milliseconds on current generation consoles.

In [19], de Melo et al. compare and analyze different 3D robotics simulators. V-Rep [20], Gazebo [21] and Unity [12] were compared, as they were found to be the most popular robotics simulators, by number of mentions and uses on *Google Scholar*. By this metric, the runner-up simulators were Webots [22] and ARGoS [23, 24]. From their survey, they determined that ROS integration, the back-end physics engine, model file compatibility (URDF, SDF), language support and software licensing were important points of evaluation. They concluded that Unity and V-Rep were superior to Gazebo in terms of user-friendliness and customizability. Still, they found that Unity generally had fewer robotics-oriented features, and provided inferior ROS integration. It should be noted that Unity has done work to provide better ROS integration and features that are useful in robotics simulation since the publication of this article [25].

Vasstein et al. use the digital-twin platform Autoferry Gemini [26] to simulate electromagnetic radiation sensors using the geometry buffer and custom shader techniques, allowing advanced sensor simulation to be performed in real-time using the GPU [27]. In this work, physically based models are used to generate realistic data from the virtual sensors, including LiDAR, infrared camera and radar. The LiDAR simulation is done by fetching depth buffers from an array of virtual cameras in a custom pass. Then, using compute shaders, the depth buffers are stitched and a 3D pointcloud is generated.

They report simulating 30 million points per second at a stable 50 frames per second. They also note that the depth buffer stitching method introduces errors in the point cloud, as cylindrical beams are approximated with regular polygons.

# 3. Foundations

In Foundations, we present the relevant theoretical background knowledge required to follow along with our implementation in the following chapter.

## 3.1. Unity

Unity is a video game engine developed by Unity Technologies [12]. Its features allow users to render both 2D and 3D graphics, simulate physics, and interact with scenes through an intuitive editor interface. Unity is generally considered to be beginner-friendly, as little knowledge about computer graphics is required to create a basic project.

Custom meshes and materials can be imported into Unity, and custom game logic and behaviour can be defined using C# scripts. By default, all scripts run on a single CPU thread, but by using Unity's Job system one can write more efficient, multi-threaded scripts with thread-safety [28].

For the most part, scripting is done through classes that inherit the `MonoBehaviour` base class. Using this base class, special methods and attributes are exposed to easily integrate custom scripts into the world. The most important of these are the `Start()`, `Update()` and `FixedUpdate()` methods. Code defined in the `Start()` method is run only once (when the game is launched), code in `Update()` is executed every frame and `FixedUpdate()` is called once per iteration of the physics loop.

### 3.1.1. Physics Engine

Unity's default 3D physics engine is based on Nvidia PhysX [29], which handles character control, collisions, rigid body physics etc. [30]. For custom physical interactions which are not accounted for by the built-in physics engine, methods of the `Rigidbody` class, such as `AddForce()` and `AddTorque()` can be invoked in user-defined scripts [31]. This is a simple yet powerful API which allows for the simulation of complex systems, such as cloth physics and fluid-to-solid interactions.

### 3.1.2. Rendering Pipeline

In Unity, 3D objects with materials are rendered to the screen automatically, through one of the rendering pipelines. The High Quality Rendering Pipeline uses physically-based rendering techniques to create visually pleasing graphics that react to lighting and geometry in a physically realistic way [13]. The look of an object can be modified by editing the material that the object uses and its associated shader. Shaders are programs that run on the Graphical Processing Unit (GPU). These programs are often related to graphics, but may also be general-purpose programs [32].

Custom shaders are written in Shaderlab, a wrapper for the High-Level Shader Language (HLSL), allowing for complete control of how an object is rendered. Custom Pass volumes can be used to inject special rendering behaviour through shaders and C# scripts. Custom passes can also be used to extract data from the rendering pipeline, such as the depth- or z-buffer. This can be useful for simulating sensors that measure distances to objects, such as LiDARs and stereo camera systems.

### 3.1.3. 3D Assets

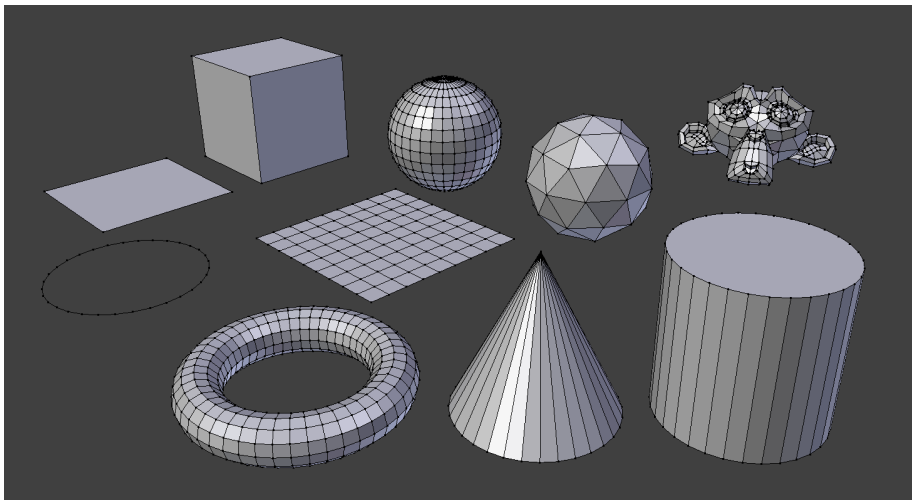


Figure 1: *This is an overview of the standard primitives included in Blender [33].*

In addition to Unity's 3D rendering capabilities, it provides some tools for generating simple 3D objects [34]. The basic shapes like a sphere, cylinder or cube make up what is called *Primitives* in modeling [35]. These usually serve as the starting point when modeling a more complex mesh, or as a placeholder level designers can use while waiting for more detailed geometry. A mesh in the field of 3D modeling is the collection of components that establish the geometry. These are vertices (vertex singular), edges and faces [36, 37]. A vertex is a position in 3D space, and the vertices of a mesh are stored as an array of coordinates. The edges are the straight lines connecting vertices that make up a face. Faces are the area between vertices. Following is an example of how a cube mesh is defined in the Wavefront OBJ format [38], `v` for vertices and `f` for faces.

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

The general term for faces are polygons, but they are further categorized as triangles (three vertices), quadrangles (four vertices) and n-gons (five or more vertices) [36]. In

3D graphics, the most useful of these polygons are the triangle, as it is inherently planar and can be applied a fixed shading algorithm. These algorithms, like the Gouraud [39] or the Blinn-Phong [40], lend themselves to be implemented in hardware [41]. Another argument for triangles are their simplicity as the smallest possible surface definition, and the fact that any polygon with more sides than three can be broken down into a triangle [42, 43, 44]. There have been invented several techniques for mitigating memory bandwidth limitations like triangle strips [42] and compression [45].

Moving on from the Primitives, we enter the domain of the 3D artist. To make more complex models, one needs an understanding of digital content creation applications like Blender [46] or 3D Studio Max [47]. These are advanced tools for artists to create meshes of high detail. Unity does provide its own simple ProBuilder tool which allows for some basic editing of meshes within the engine itself [48].

The geometry itself is only half of the puzzle. We also need colors to draw the geometry to the screen. As it happens, the vertices of the mesh define these properties, holding potentially color information, texture coordinates and normal (facing) direction [49].

In 3D art, one way of coloring assets is by mapping a detailed image onto the surface of the mesh. This can depict surface detail not defined by the geometry itself [50], and requires connecting the 3D mesh to 2D space [51]. This process is called UV mapping, and results in the vertices carrying two sets of positions, its regular 3D location and a new UV coordinate specifying its location in 2D texture space. It is called UV coordinates because it is named after the axis of a texture, V in the vertical and U in the horizontal, starting at (0,0) in the top left corner, and ending at (1,1) in the bottom right [52].

## 3.2. Robot Operating System

The Robot Operating System (ROS) [53] is a software framework for developing robotics applications. The framework is designed to be hardware-agnostic with an emphasis on modularity. Through open-source ROS *packages*, developers avoid the complexity of writing algorithms from scratch, instead opting for custom configuration and parameter tweaking of pre-existing packages. Packages are groups of *nodes*, which are pieces of code with different specific tasks. Nodes can be reused on different robotic systems. For example, a given ROS stack might have a dedicated `sensors` package, including individual drivers and processing nodes for various physical sensors, e.g. cameras, inertial sensors or odometry sensors.

*Topics* and *messages* are core concepts in ROS, and are the main forms of communication between nodes. The communication pattern used is that of "publish-subscribe", or pub-sub. Nodes may publish (transmit) or subscribe (receive) messages on *topics*, which are messaging channels that have a defined message type. *Messages* are packets of information with a standardized structure that are sent across topics. For instance, a camera driver may publish messages of the type `sensor_msgs/Image` on an image topic. It's the standardization of data transfer and message types that allow for packages to be hardware-agnostic. For example, a ROS image processing package should not depend on the specific physical camera a robot is equipped with, as long as a sensor driver is publishing `sensor_msgs/Image` messages on the configured topic.

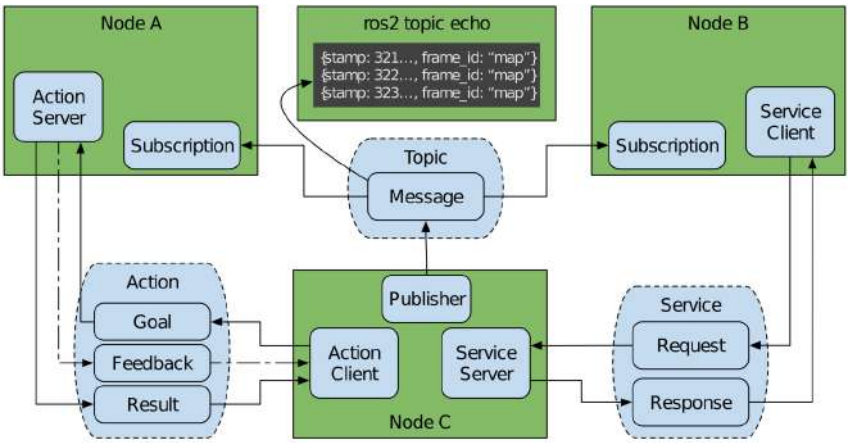



Figure 2: *The diagram shows the various ways nodes communicate through topics, actions and services [54]. The ROS command-line-interface allows for introspection, like monitoring messaging traffic or manually publishing messages.*

ROS also supports *Actions* and *Services*, which allow for more complex behaviour. Both services and actions consist of servers and clients. Services in ROS are synchronous. This means that after a client calls on a service by sending a request to the server, it halts execution until the server has responded. This is useful for operations where a response is needed before continuing, such as getting the robot's current position or checking if a sensor is activated.

Actions are better suited for long-running tasks. They are asynchronous, allowing a client to send a goal to an action server and then continue operating without waiting for the result. Clients can monitor the progress of the action and can cancel the action if needed. This is particularly useful for tasks like moving a robot to a target location, where updates or changes might be required along the way.

Originally, ROS used the Transmission Control Protocol (TCP) [55] as its standard topic transport protocol. In ROS 2 however, a Data Distribution Service (DDS) [56] implementation is used.

## 3.3. Sensors

Simulated sensors are fundamentally different than real, physical sensors. Still, it is valuable to understand the underlying principles and theory of the sensors, as this knowledge can enhance the realism of the simulation data.

### 3.3.1. Light Detection and Ranging

Light Detection and Ranging (LiDAR) sensors are used to measure distances to surrounding objects using the ToF (Time of Flight) measuring principle. These sensors emit highly focused beams of light that scatter off of surfaces in the environment. The reflected light is detected by the sensor, and the ToF of the emission is used to calculate the distance to the object. The working principle of LiDAR sensors is similar to that of Radar (Radio Detection and Ranging), except that LiDAR uses beams of light instead of radio waves [57].

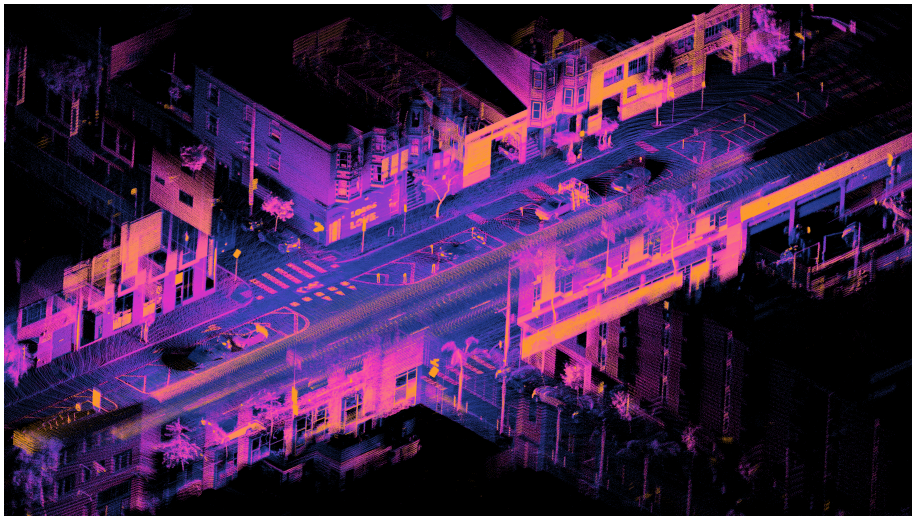


Figure 3: *A point cloud produced by an Ouster 3D LiDAR sensor, taken from [58].*

LiDAR sensors are often categorized into two types: 2D and 3D. 2D LiDAR sensors perform range measurements along a 2D plane (typically the horizontal plane), where measurements are taken with regular angle increments. This produces a planar point cloud, which can be used for 2D mapping, navigation and obstacle avoidance [59]. 3D LiDAR sensors can also emit light along the vertical direction, producing 3D point clouds. In recent years, algorithms have been developed to perform segmentation and object detection on raw point clouds produced by 3D LiDAR sensors, such as the PointNet neural network [60].

### 3.3.2. Depth Cameras

Depth cameras are a class of sensors that use various techniques to produce 2D depth images [61]. One method for estimating the depth of a scene is through "computer stereo vision". This technique uses two individual RGB cameras, separated by a known distance referred to as the baseline. By synchronizing the two cameras, pairs of images are captured and a process known as "stereo matching" is performed [62]. The result of the stereo matching process is a set of feature correspondences between the two images. These correspondences allow for the computation of a depth map, which represents the distance from the camera to the captured object.

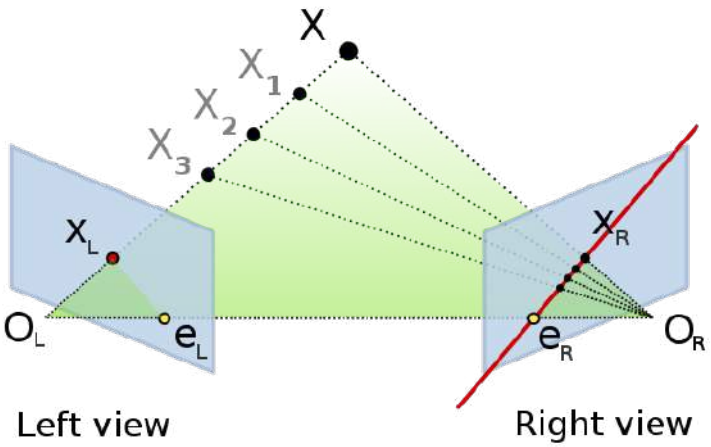



Figure 4: *The figure shows how the ray projected by a pixel in the left view corresponds to a line in the image plane of the right view. This property comes from epipolar geometry, and is exploited in the stereo matching procedure [63].*

To perform depth estimation via the stereo vision principle, the baseline and focal lengths of the cameras must be known. The identification of these parameters is referred to as "camera calibration". Successful stereo depth estimation may also require image distortion correction, compensating for warping effects and perspective distortion [62].

Depth cameras may also use the ToF sensing principle to generate depth images [64]. This method of capturing depth uses the same principle as the LiDAR sensor, where light rays are emitted and their travel times are measured to compute a distance.

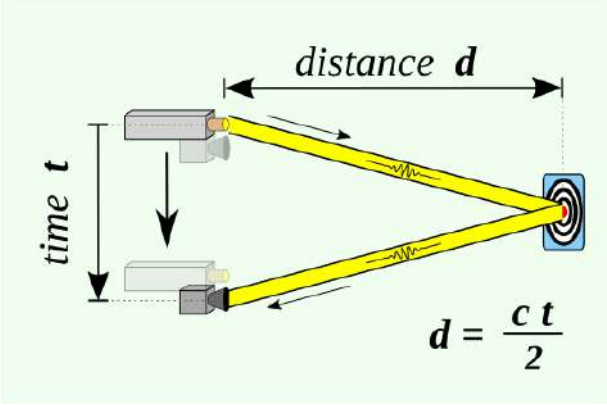



Figure 5: *Time of Flight camera sensing principle [65].*

"Structured Light Projection" may also be used to generate depth images. This technique involves emitting a known light pattern onto the scene and analyzing the resulting projection of the pattern with a camera. The depth image is generated by evaluating the deformation of the emitted light pattern [66].

This depth map can be represented as a grayscale 2D raster image, where the intensity of each pixel represents the distance from the camera to the object.

### 3.4. Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) refers to the problem of generating a map of an unknown environment while determining one's position on that map [67]. This makes SLAM particularly useful in environments where external tracking systems, such as Global Navigation Satellite Systems (GNSS), are unavailable or unreliable. The output of a SLAM algorithm may be used to inform safe and efficient autonomous planning and navigation.

There are many ways to tackle the SLAM problem, and it is still an active field of research. The particle filter is a popular technique for solving the "online" SLAM problem, where the estimated map and localization are continuously updated during operation. An implementation of this technique can be found in OpenSLAM's GMapping algorithm, which uses odometry data and laser scans typically, obtained from 2D LiDAR sensors [68]. The sensors required for this algorithm are relatively low-cost, and GMapping became the de facto standard SLAM solver for robots running ROS, though it was replaced by `slam_toolbox` in ROS 2 [69].


Figure 6: *An example of successful loop closure, where visual features from new and previous data are matched [70, 71].*

Loop closure is an important part of the SLAM algorithm, and refers to the process of associating recent sensor data to data that has been previously observed [72]. The loop closure technique may be applied to different sensor types, for both 2D and 3D SLAM. In 2D applications, loop closure may be applied to the LiDAR data, where recent scans are matched with previous scans with similar geometric features. Similarly, in the context of visual slam, loop closures occur as new images are successfully matched with previously captured images.

### 3.5. Thrust Allocation

Thrust allocation attempts to find a set of individual actuator commands that satisfy some total action on the vehicle, where dynamics and actuator constraints may be included. In the context of autonomous vessels, a thrust allocation scheme is required to convert a generalized force vector (typically produced by automatic control) into a set of angles and thrust magnitudes [73]. These thrust vectors may then be forwarded to the propulsion hardware, such that the desired total force is applied to the vessel.

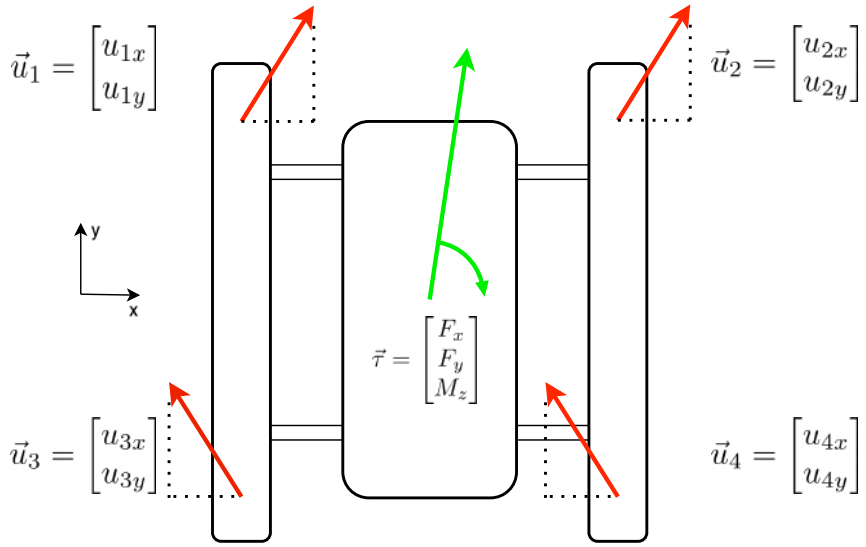



Figure 7: *An illustrative example of a thrust allocation solution. The desired generalized force vector $\vec{\tau}$ is achieved through the individual thrust vectors $\{u_1, ...u_4\}$, expressed in Cartesian coordinates.*

To solve the thrust allocation problem, a constrained optimization problem is formulated. The problem statement depends on the specific requirements of the allocation scheme, which might include objectives such as minimizing energy consumption and actuator wear-and-tear.

A general formulation of the thrust allocation optimization problem is as follows:

$$\min_{\vec{u}} G(\vec{u})$$

subject to: equality and inequality constraints

Where $\vec{u}$ is a vector containing the individual thruster commands. A simple case of this optimization problem is one where $G(\vec{u}) = \vec{u}^T\vec{u}$, and a single equality constraint is included: $\vec{\tau} = B\vec{u}$. Here, $\vec{\tau}$ is the generalized force input vector in Cartesian coordinates, and $B$ is the thruster configuration matrix [74]. Intuitively, solving this optimization problem amounts to finding the lowest-energy propulsion command $\vec{u}$ that produces the desired total force on the vehicle $\vec{\tau}$. The elements in $\vec{u}$ are force components expressed in Cartesian coordinates instead of in polar coordinates (thrust and angle). This is done to avoid non-convexity in the optimization problem, as the $B$ matrix would include trigonometric functions.

### 3.6. Hydrostatics

Hydrostatics, buoyancy, describes the force that is applied to an object at rest in fluid [75]. The buoyant force exists due to the properties of hydrostatic pressure. It is called 'static' because it is not dependent on the object's motion relative to the fluid, such as currents or propulsion. This pressure increase with depth below the surface, and this creates more pressure on a submerged object's deepest point than on its shallowest [76, p. 30].

Pressure is applied to the surface's normals, if the fluid is at rest [76, p. 30]. For a multifaceted body, pressure is applied in all its directions of contact with the fluid, the magnitude of this pressure being greater on its deepest parts. The effect in practice is that the horizontal forces cancel out, and we are left with a net force in the vertical axis with direction opposite that of gravity [75]. This force is described by the principle of Archimedes (287 BC - 212 BC), which states that

> "... any body completely or partially submerged in a fluid (gas or liquid) at rest is acted upon by an upward, or buoyant, force, the magnitude of which is equal to the weight of the fluid displaced by the body" [77].

Archimedes' principle is represented by this equation,

$$F_{\mathrm{b}} = \rho \mathrm{g} V \tag{1}$$

where $F_{\mathrm{b}}$ is the buoyancy force, $\rho$ is the density of the fluid, g is the scalar force of gravity and $V$ is the volume of the submerged object.

As claimed by Jacques Kerner [78], there are two ways to calculate buoyancy forces, a surface-based approach and a volumetric approach. In our simulator, we have implemented both, detailed in the *Implementation* section.

### 3.7. Hydrodynamics

It was Daniel Bernoulli (1700 - 1783) who first coined the term hydrodynamics [76, p. 202] with his book *Hydrodynamica* published in 1738 [79]. As a branch of physics, hydrodynamics is the study of the motion of fluids and the forces on solid bodies immersed in and moving relative to these fluids [80]. Bernoulli, in addition to Leonhard Euler (1707 - 1783), is credited as the founders of modern hydrodynamics [81]. In *Hydrodynamica*, Bernoulli did the first attempt at applying a general dynamic principle to fluid motion [82]. Daniel Bernoulli and his father Johann (1667 - 1748) used the *vis viva* principle of Gottfried Wilhelm Leibniz (1646 - 1716), an early formulation for conservation of energy [82, p. 5]. The statement of conservation of energy is that energy cannot be neither created nor destroyed, only transferred or transformed [83].

With this in mind, Bernoulli deducted that an increase in the dynamic pressure (an increase in kinetic energy), which is comprised of half density $\rho$ times velocity $v$ squared, would lead to a decrease in static pressure $p_{\mathrm{s}}$ (potential energy). It follows then that the sum of these pressures are a constant total pressure $p_{\mathrm{t}}$ throughout the flow [84]. Bernoulli's equation is given by:

$$p_{\mathrm{s}} + \frac{\rho v^2}{2} = p_{\mathrm{t}} \tag{2}$$

There are limitations to the applicability and validity of Bernoulli's equation, as he himself was aware. He knew some energy was lost to turbulence. Further, he knew that his hypothesis built on parallel slices would be proven only for narrow vessels with gradually variable sections [82, p. 9]. With its shortcomings, Bernoulli's theorem can be said to be valid only when the flow is *isentropic* (frictionless work transfer) and steady (unchanging over time) [85, p. 159]. One of the notable tools enabled by Bernoulli is the Henri Pitot (1695 - 1771) tube for measuring air velocity [84], [86, Ch. 14.4].

It was Euler, a student of Johann Bernoulli, who first produced the dynamical equations of motion of a rigid body [86, Ch. 13, eq. 13.6 and 13.7]. Euler paid tribute to the Bernoullis and Jean-Baptiste le Rond d'Alembert (1717 - 1783), all of whom anticipated some of the essential features of Euler's approach [82, pp. 25-26]. Whereas the equations of Bernoulli was more geometric and case-specific, Euler adapted them into a more general analytic form [82, p. 4]. He achieved his equations in a clear and modern form, unlike his predecessors [82, p. 25]. Here is Euler's equations of inviscid motion:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\frac{\nabla P}{\rho} \tag{3}$$

where $u$ is the fluid velocity, $P$ is the pressure, and $\rho$ is the fluid density [87]. The $\nabla$ is the vector differential operator.

In the early days, fluid was treated as *inviscid* (where viscosity approaches zero, i.e. negligible). This was the case with the equations of both Bernoulli and Euler. It was Claude-Louis Navier (1785 - 1836) that first contended with this variable of viscosity, based on his general theory of elasticity. Viscosity describes a fluid's resistance to flow or to change shape [88], compare for instance honey and water. Navier gained little contemporary recognition for his hydrodynamic equation, and it was rediscovered by Augustin-Louis Cauchy (1789 - 1857), Siméon Denis Poisson (1781 - 1840), Adhémar Saint-Venant (1797 - 1886) and lastly George Gabriel Stokes (1819 -1903), all of whom made their contributions to how it is presently formulated [82, p. 101]. The Navier-Stokes equation of incompressible fluid flow:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = -\frac{\nabla P}{\rho} + \nu \nabla^2 \vec{u} \tag{4}$$

with the same parameters as Euler's equation, and otherwise similar if not for the last term, which is where $\nu$ is introduced as the kinematic viscosity [89].

Experiments with small model ships by William Froude (1810-1879) in the fall of 1867, made him conclude that the wave patterns generated were significantly related to ship resistance, and represented an energy loss [76, p. 204]. The waves are caused by the rigid body displacing water as it moves, expending some forces not wasted when at rest. In addition to this wave making displacement resistance, Froude wanted to determine the frictional resistance of water molecules being dragged along the hull [76, p. 204]. Through his experiments, he discovered several important variables contributing to resistance, mainly velocity, surface area and smoothness, viscosity and fluid density. Froude contributed vitally to highlight the usefulness of models for testing ship performance through his formulation of *corresponding speeds* [76, p. 204]; [82, p. 282]. This describes a relationship between attainable speed and scale, formulated $V/\sqrt{L}$, with $V$ for velocity and $L$ for length.

Osborne Reynolds (1842 - 1912) contributed to the field of hydrodynamics with his Reynold's number, a dimensionless quantity describing the flow behaviour of fluid [90]. The number express the ratio of *inertial* (resistant to change or motion) to viscous forces [91]. He made experiments with a tube fed with water from a large reservoir, and added dye to the currents to visualize the flow. Controlling the speed of the water, Reynolds observed the following. At low velocity, the dye strands remained stable and flowed along the streamlines, parallel to the tube walls, with maximum velocity

at the center. This he called *laminar* flow. When the velocity was increased, the dye streaks became first sinous before breaking up completely. The phenomenon was named *turbulent* flow [76, p. 214]. Here is the equation for Reynold's number:

$$\mathrm{Re} = \frac{\rho v l}{\mu} \tag{5}$$

where $\rho$ is density as usual, $v$ is the characteristic velocity, $l$ is characteristic length and $\mu$ is the dynamic viscosity coefficient [91].

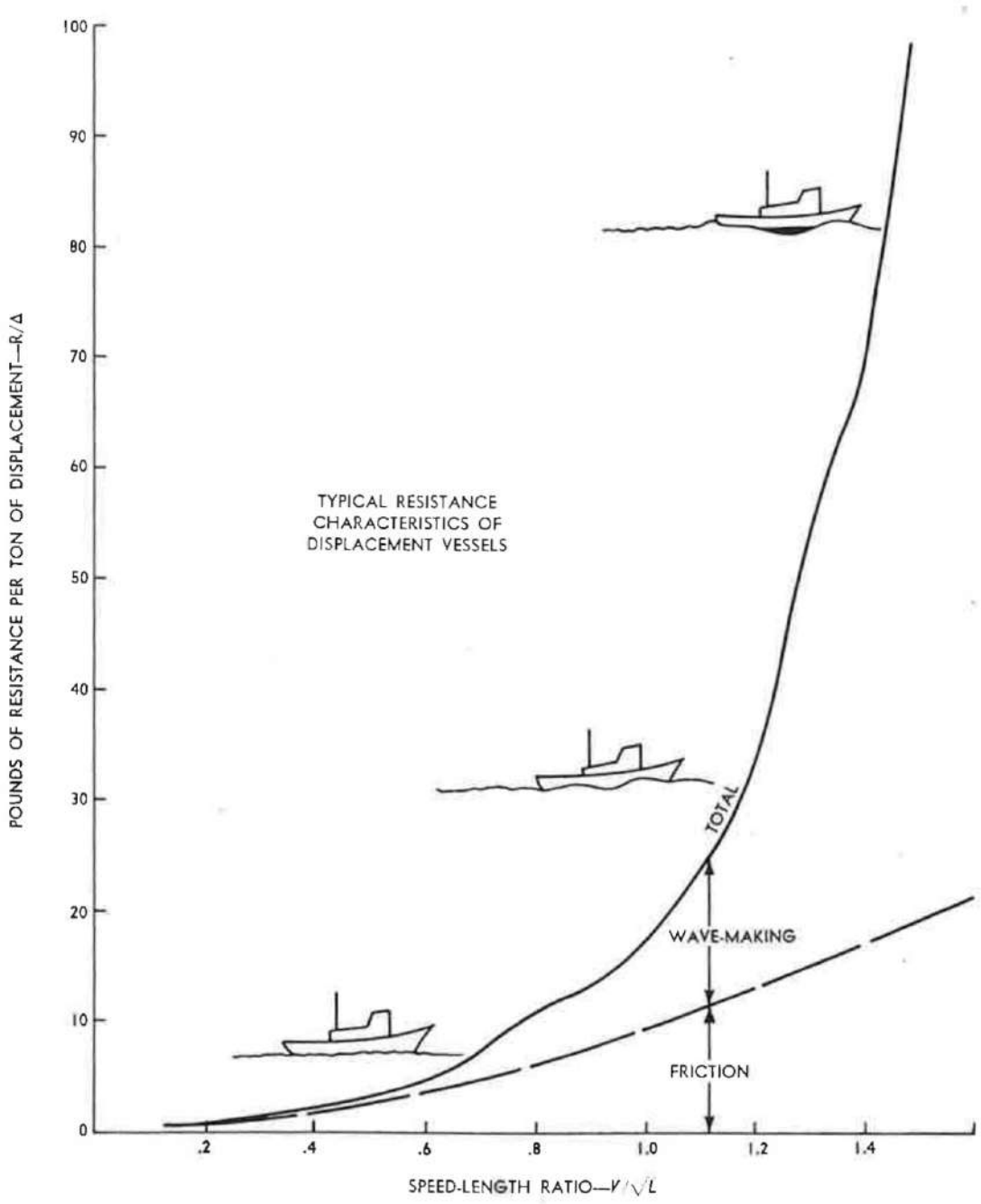



Figure 8: *Original caption: "Resistance versus $V/\sqrt{L}$ in a characteristic speed-power curve for a displacement ship (note the ship-wave profiles)" [76, p. 209, Fig. 11-3].*

The combination of the Euler and Navier-Stokes equations produce a more complete set for investigating the stability, motions and trajectories of the rigid body [86, Ch. 13.1]. When dealing with a rigid body, it is necessary with two coordinate systems, one representing the world, and one attached to the body [86, Ch. 13.3]. The world axis is centered on zero, and the body axis is centered on the point G, which is its center of mass.

There are several resistances to steady forward motion in fluid that contribute to energy loss. Directly quoted from [92] (seemingly inspired by [93, p. 193]), these are:

1. Friction between the water and the hull surfaces
2. Energy expended in creating the wave system caused by the hull
3. Energy put into eddies shed by the hull and its appendages (e.g., the rudder)
4. Resistance by the air to above-water parts of the ship

To begin with the resistance caused by friction between the water and the hull surfaces, we know that Froude demonstrated this with his model experiments, and together with his son Robert Edmund (1846 - 1924) they created the following expression [76, p. 205, eq. 11-2], [93, p. 194], [94, p. 7]:

$$R_f = fSV^n \tag{6}$$

where $R_f$ is the frictional resistance of the fluid (Newtons), $S$ is the wetted surface ($m^2$), $V$ is velocity ($^{m}/_{s}$) and the friction coefficient $f$ and exponent $n$ depend on the length and nature of the surface, defined by Froude through experiments. Froude found $n$ to usually have a value of around 2, a little less for smooth planks [76, p. 205]. We here have the significant variables that determine magnitude of frictional resistance, that is, submerged surface area and flow relative velocity.

As seen in Figure 8, the wave-making resistance is consequential in defining the upper limit of ship speed [76, p. 208]. The bow and the stern are the wave pattern's main points of origin. The wave-making also creates a pressure difference, with the bow experiencing higher and the stern lower pressure [94, p. 3]. Waves moving in lockstep with the ship is called transverse, and their crests (tops) and troughs (bottoms) are perpendicular to the ships course vector [76, p. 208]. The phase (length between crests) of transverse waves is a function of the phase velocity. Facinatingly, this phase velocity is the same as ship velocity and we get an equation for the critical speed-length ratio [76, p. 209, eq. 11-15]:

$$V_s/\sqrt{L_s} = 1.34 \tag{7}$$

where $V_s$ is ship velocity and $L_s$ is ship length. When a surface ship exeeds the speed length ratio of 1.34, is starts climbing its bow wave, and the resistance increases prohibitively [76, p. 210]. The energy expended by the rigid body in its motion aganst the waves are equal to the energy required to maintain such a wave system [94, p. 15].

More residural resistance and hence energy loss is found in eddies, or vortices, resulting from the creation of turbulence and some air resistance for big ships. We should distinguish this type of eddy creation from the aforementioned frictional resistance, as that too creates eddies due to tangential skin friction making a turbulent belt around the body [94, p. 27]. The eddies generated by frictional resistance are due to the continuous shearing (sliding against one another) between the body and the fluid. The other type of eddies are driven by flow separation due to abrupt changes of form or appendages [94, pp. 27-28]. Flow separation is also called boundary separation because

it describes the phenomenon of boundary layer detachment from a surface and into its wake. This is always associated with the formation of vortices and accompanying energy loss [95, p. 24]. The concept of a boundary layer was first introduced by Ludwig Prandtl (1875 – 1953) in 1904 [96]. According to Prandtl's theory, at high Reynolds numbers, the influence of viscosity is restricted to a very thin layer near the solid wall [86, Ch. 5.7], [95, p. 23]. When the boundary layer can no longer adhere to the surface due to conflicting pressure gradients, flow separation occurs, leading to increased drag and flow instability [97, p. 405].

Figure 9: *Original caption: "**Kármán vortex street behind a circular cylinder at $R = 105$.** The initially spreading wake shown opposite develops into the two parallel rows of staggered vortices that von Kármán's invicid theory shows to be stable when the ratio of width to streamwise spacing is 0.28. Streaklines are shown by electrolytic precipitation in water.* Photograph by Sadatoshi Taneda*" [98, p. 57, Fig. 96].*

# 4. Implementation

In this section, details about the implementation of the digital twin & simulation platform are presented.

## 4.1. System Architecture

At a high level, the software system consists of two main parts: the simulation (Unity) and the development environment (ROS 2). It's important to consider software compatibility and system requirements, as a well-designed system architecture reduces friction in the adoption of the proposed platform. The goal is for everything to be easy to set up and to work smoothly on *most* developer setups. We mainly focus on compatibility with Linux (Ubuntu) and Microsoft Windows workflows, though the proposed architecture may also be compatible with other operating systems.

The simulation is packaged as source code which can be directly launched through the Unity editor/hub. This means that it can be run on any machine supported by Unity with HDRP. Throughout the project, the simulator has been tested and developed on Windows 11, Apple MacOS Sonoma and Ubuntu (22.04 LTS).

For ROS 2 interoperability, there are some OS compatibility challenges, as illustrated in the following subsections.

### 4.1.1. Ubuntu

For machines using Ubuntu 22.04, the setup is straight forward, not requiring any containerization or virtualization. The simulator can be run as a standalone Linux executable or in the Unity Editor, and will automatically publish data to the ROS 2 network. The setup of the ROS 2 development environment is also straight forward, as ROS is targeted at Ubuntu [54]. Given a valid IP and port number configuration, the developer may then subscribe to the ROS 2 data topics as usual.

### 4.1.2. Windows option 1 - Docker

As ROS 2 is not primarily targeted at Windows platforms, some additional setup is required for Windows 11. It should be noted that this is not a consequence of our specific system, but a fact of ROS 2 development on Windows in general. A common solution to this problem is to run ROS 2 on a virtual Ubuntu machine, or through a containerized environment. Two working solutions were tested, varying in complexity, overhead and flexibility: Docker [99] and Robostack [100]. An overview of this software setup and the general system architecture is shown in Figure 10.

Using Docker, we containerize Ubuntu 22.04 with ROS 2, enabling full development capabilities on Windows platforms. To enable applications with graphical user interfaces (GUIs) such as RViz2 [101], we use VcXsrv [102], which forwards forward display data through the containerization layer.

This solution delivers a fully functioning setup for Windows 11 machines. The main disadvantage of this approach is the added complexity of working with Docker containers, and the consequent performance overhead. The added overhead is particularly noticeable when GUI applications are run and forwarded through the container.

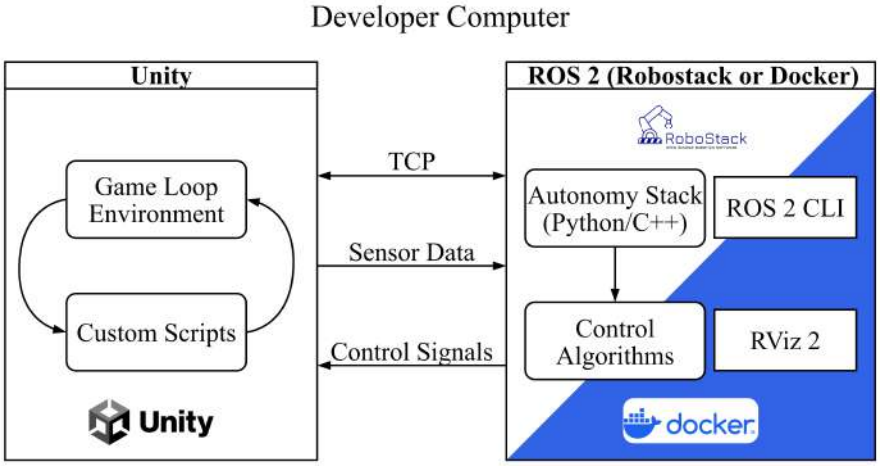



Figure 10: *Simulation and development environment setup for a Windows machine*

### 4.1.3. Windows option 2 - Robostack & Conda

Robostack offers a more "native" solution for ROS 2 development on Windows. Instead of using containers, it leverages virtual environments (Conda) to reproduce a ROS environment on Windows [103]. With this solution, display forwarding is not necessary, and we constrain the setup to a single operating system. GUI applications like RViz run natively, and the development of ROS 2 software is simplified, as we are not working through a container interface. The disadvantage of the Robostack approach is that not all package binaries are available yet. However, new packages are being ported every day, and the Robostack community is actively adding packages per user request.

## 4.2. Physics & Modelling

Our strategy for implementing physics aims to achieve visually appealing responses while ensuring an adequate degree of physical accuracy. Balancing physical correctness, implementation complexity and usability poses a significant challenge, especially in the presence of real-time constraints. Since different users have different priorities and requirements, we provide a modular physics system, where the user may pick and choose between approaches for their specific use case. For instance, if a user is developing a neural network for collision avoidance logic or evaluating an object detection model, they can opt to replace complex buoyancy and hydrodynamics algorithms with simplified or proxy physics. This facilitates enhanced performance for specific use cases by focusing computational resources on the areas most critical to the user's objectives.

The back-end physics engine used in Unity is an integration of the Nvidia PhysX engine [30]. Using this built-in physics engine, we can directly apply the calculated forces and moments on the floating vessels using the associated API.

The first major challenge in simulating the response of a vessel in ocean waves is acquiring accurate information about the state of the surrounding water. In particular, we need information about the amplitude and velocity of the wave field in a neighbourhood around each simulated vessel. This is essential for calculating the various forces to exert on a vessel as it traverses the ocean. The HDRP water system provides an API

for querying the height of the water at discrete points, and allows the user to set environmental properties such as distant wind speed, current speed and local water ripples. Using the provided API we are able to get an approximation of the surrounding wave field, such that hydrostatic and hydrodynamic effects can be simulated.

### 4.2.1. Water Interaction Pipeline & Vessel Representation

To calculate buoyancy and various hydrodynamic forces, we adopt Jacques Kerner's approach [78], relying heavily on this in the following Sections 4.2.2., 4.2.4. and 4.2.6.. In this approach, the 3D triangle mesh representation of the vessel provides the foundation for how these forces are determined. This is practical, since it conforms to how 3D objects are generally presented in video game engines like Unity. On the other hand, it introduces an intrinsic approximation, as the 3D mesh representation discretizes a continuous geometry into a set of faces and vertices. The fidelity of this discretization imposes a trade-off between realism and computational costs. For this reason, it is common to store two separate meshes of the target vessel: a high fidelity mesh that is used for rendering purposes and a low-fidelity, simplified mesh for the evaluation of physical forces.

The general process of determining the hydrostatic and hydrodynamic forces can be summarized as follows:

1. A piecewise linear height map of the neighbouring water surface is generated by sampling the water height at discrete points
2. The height map and the simplified physics mesh are used to determine the part of the vessel that is completely submerged.
3. The buoyancy force on the vessel is calculated based on the completely submerged geometry.
4. The hydrodynamic forces are calculated globally via a state-space model, or locally for each surface element.
5. The total hydrostatic and hydrodynamic forces are applied to the rigid body via the Unity physics engine.

### 4.2.2. Water Surface Approximation

Determining the part of the vessel that is submerged in water amounts to checking the height of the water at different points on the vessel's surface geometry. This can be done using the provided HDRP Water System API directly, which uses a displacement method to evaluate the water height at a given point [104]. However, this method uses an iterative algorithm to determine the water height, which becomes computationally expensive for large meshes. Instead of using this API for every mesh vertex, we interpolate over a fixed grid of samples.

First, a regular grid of points centered at the target vessel position is generated. We refer to this object as the "water patch". The size of the water patch is such that the projection of the vessel is fully contained within it. The resolution $n$ of the patch (the number of rows/columns) determines how well the water patch approximates the actual neighbouring wave field. The height of each point in the water patch is set to the height of the water at that point, obtained through the previously discussed querying API.

For each subdivision of the water patch, two triangle plane equations are obtained. For instance, a water patch with $n = 4$ will have $(4 + 1) \cdot (4 + 1) = 25$ subdivisions, resulting in a total of 50 triangle plane equations. With these plane equations, a cheap method for water height querying is obtained. Figure 11 shows the result of this process for a patch with $n = 9$. Each floating object is allocated one such patch, horizontally centered around the object.

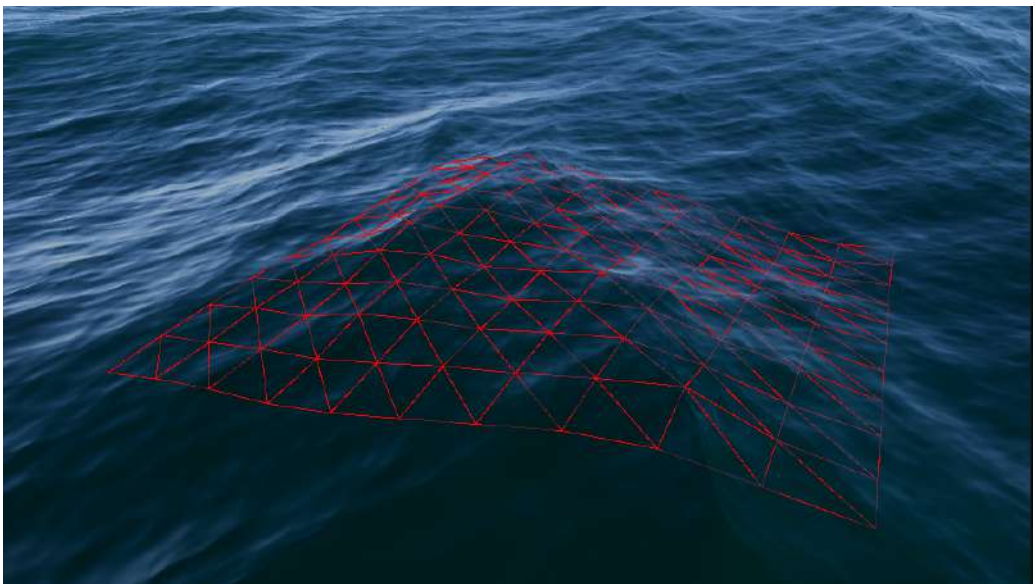


Figure 11: *Visualization of the water patch approximating the shape of the water surface. This instance has $n = 9$, resulting in 200 triangle plane equations.*

### 4.2.3. Determining the submerged sub-mesh

Once the water patch has been updated, the vertices of the target mesh are iterated over, and the water-relative depth of each vertex $y_i$ is calculated using the patch approximation. A vertex $i$ is considered submerged if $y_i \leq 0$. Triangles whose vertices are all submerged are added to a list of totally submerged triangles. When only one or two vertices are submerged, a heuristic triangle-cutting algorithm is performed to further refine the approximation. The details of the triangle-cutting algorithm is presented in [78]. This algorithm also approximates the water-line of the vessel, which can be useful for visual effects.

Figure 12: *The result of the submersion algorithm. Fully submerged mesh triangles are highlighted by green lines. The water patch is not shown here.*

The result of this process is a set of triangles that are considered totally submerged, i.e we obtain a fully submerged sub-mesh which can be analyzed on its own. This set is then used to calculate the hydrostatic and hydrodynamic forces.

### 4.2.4. Hydrostatics - Triangle Model

To simulate hydrostatic forces, we calculate the total displaced volume and the corresponding centroid (the center of buoyancy). The buoyancy force is then calculated by Archimedes' principle and applied at the object's center of buoyancy. To compute the displacement volume we sum up volume contributions from each submerged triangle. This approach is a CPU-based adoption of [18].

The displaced volume contribution from a single submerged triangle $i$ is obtained by

$$V_i = -\text{sgn}(n_y) A_h d(\vec{r_i}) \tag{8}$$

where $n_y$ is the vertical component of the outward-pointing triangle normal, $A_h$ is the area of the horizontal projection of the triangle and $d(r_i)$ is the water depth at triangle position $r_i$. Visually, this volume is a triangular prism of water, extending upwards from the submerged triangle face to the surface of the water.

$-\text{sgn}(n_y)$ ensures that downward-pointing triangles contribute positively to the volume displaced volume, and vice versa. The total volume displaced is then simply:

$$V = \Sigma_i V_i \tag{9}$$

Geometrically, this sum "carves out" the volume between the submerged part of the object and the water surface.

A similar strategy is used to compute the center of buoyancy. For each submerged triangle $i$, the centroid of its water column is computed by:

$$\vec{C_i} = \vec{r_i} + \frac{1}{2}[0, d(\vec{r_i}), 0] \tag{10}$$

Finally, we calculate the total centroid of the submerged triangles combined, keeping in mind the positive/negative contributions based on triangle normal direction. This gives

us the vessel's center of buoyancy $C_b$:

$$\vec{C_b} = \frac{\Sigma_i \vec{C_i} V_i}{V} \tag{11}$$

#### 4.2.5. Hydrostatics - Voxel Model

We wanted to, for the sake of supplement and verification, have another method for calculating and applying buoyancy to our models. Based on the foundational physics formula of Archimedes' principle (1), we have constructed a simple system for evaluating this force. In this system, we use the concept of a *voxel*. The name voxel, an abbreviation, was likely first introduced in the medical field with Computed Tomography (CT) scans; the method of recreating the body of a patient in three dimensions by use of X-Ray images [105]. Voxel is short for a volume pixel, according to O. Kaleniuk [106]. Pixel is a contraction of picture element [107]. According to R. F. Graf, a picture element is the smallest distinguishable area of an image. Today we are used to the concept of pixels to describe the resolution, and thereby fidelity, of our displays and digital cameras.

In a similar way, we can think of voxels as a way to describe the resolution of a 3D model. The voxel approximation of a mesh model shares a conceptual similarity with the Riemann integral [108, Ch. 14], in that it attempts to break down a complex continuous entity (curve or mesh) into discrete parts (areas or volume elements). In both cases, the accuracy of the approximation increases as the size of the discrete parts decrease. In other words, the smaller the cubes, the more the voxel collection will look like the original mesh, as seen in [109, Fig. 14.22].

We can use this to our advantage. Having a lower resolution approximation with simple cubes is a great springboard for the calculation of physics, given its computational efficiency and easier analytical comparison. There is a practical trade-off to be determined between the accuracy of approximation, and the compute time required to execute the physics. For our case, we do not need the most detailed approximation, rather prioritizing runtime performance to provide a real-time interactable simulation. Given the fact that we only need to determine the voxelized mesh once, our buoyancy system is divided into two C# scripts: the voxelize mesh script and the voxelized buoyancy script.

##### 4.2.5.1. The Voxelize mesh script

The first script is the preparatory stage that computes the voxel approximation for the mesh. To do this, we begin by forming the bounding box from the target mesh. Within this box, we place points evenly spaced, beginning at its center, and moving outwards in x, y, and z directions. The distance between these points are equal to the edges of the voxel cubes, and are user definable.

After this process we have an evenly distributed point cloud. To make this point cloud into a voxel approximation of the target hull, we need to determine which points are inside the mesh. To do this, we iterate over each point and cast a ray in the direction of the bounding box center. The reason this works is due to a Unity quirk in regards to ray-casting; Rays that originate from inside a collider mesh, do not intersect

with this collider. Our implementation results in all the points outside of the mesh registering an intersection, while the ones inside will not.

This allows for logic to be done with the two states of ray output. We simply discard all off the points that are outside of the mesh. For the points inside, we add their position to our list of three dimensional vectors. We now have our crucial data which will be used in the buoyancy calculation script.

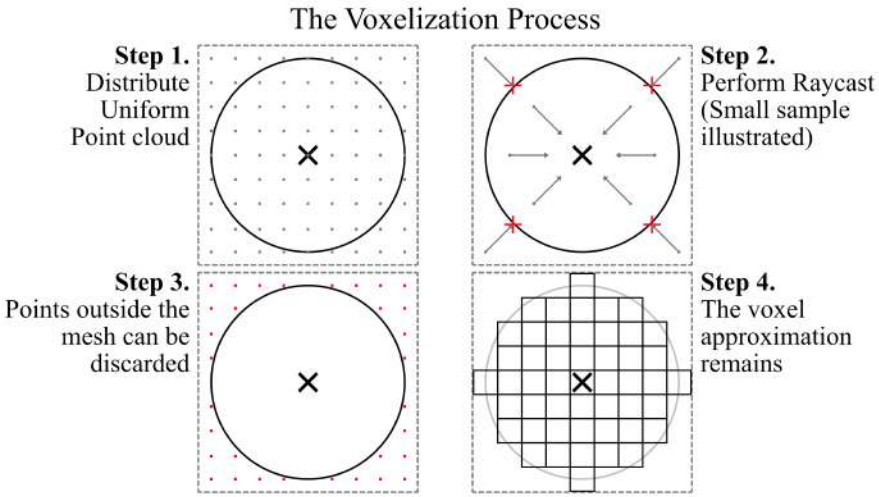



Figure 13: *Illustration of the voxelization process. Notice how the voxels here are a crude approximation of the circle at the edge cases.*

The final part of the script provides a debug visual that draws wireframes around the bounding box (red) and the voxels themselves (green).

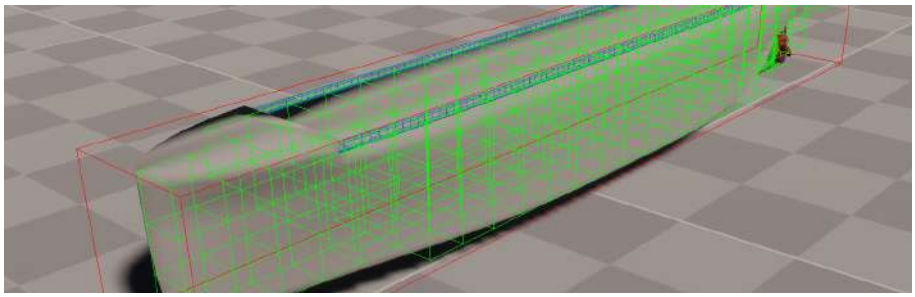


Figure 14: *An image of our voxel approximation with 5 meter cubes of a 3D mesh model.*

Because this script is supposed to be run in the Unity editor as part of an asset preparation step, we do not by default have access to this list. We therefore write the list to a JSON file, to be read whenever we please.

With our data prepared, we move over to the practical part of the system; the buoyancy script.

#### 4.2.5.2. The voxelized buoyancy script

In the buoyancy script, we begin by importing our data from the JSON file. After that, in the update function, we update our points global position to align with the transform of the target mesh. This is done easily with Unitys built-in function *TransformPoint* [110]. Each of our voxels now have global coordinates, and are added to a new list, used for querying the water height. With our water patch system, we can obtain the height distance of each point from the water surface. This relative height parameter can be used to improve the precision of our volume approximation. We can cut the height of the voxels by this distance; The result is a total height that is more than the radius if the voxel is below the water level, and less than the radius if above. We define the range in which we do this cutting procedure as having to be within the radius of a voxel, either above or below the surface. Voxels at a greater distance from the surface than this is either fully submerged or emerged. See Figure 15. for an overview of the concept in two dimensions.
We use the following formula to calculate the volume of the semi-submerged voxels:

$$V_i = \begin{cases} 0 & \text{if } h \geq \frac{l}{2} \\ l^2(\frac{l}{2} - h) & \text{if } -\frac{l}{2} < h < \frac{l}{2} \\ l^3 & \text{if } h \leq -\frac{l}{2} \end{cases}$$

Where $V_i$ is the volume for a voxel $i$, $l$ is the length of the voxels edges, and $h$ is the height of the voxel center relative to the water patch.

We sum up all the volumes of the semi-submerged voxels, and add to that the volume of the fully submerged ones.

$$V_T = \sum_{i=0}^{N} V_i$$

Here $V_T$ is the total submerged volume and $N$ is the amount of voxels submerged.

The amount of force is calculated by the formula of Archimedes (1). Finally, we calculate the center of the points by finding the average of all the points below or at water surface. Our buoyancy force is applied upwards at this center position relative to the RigidBody component.

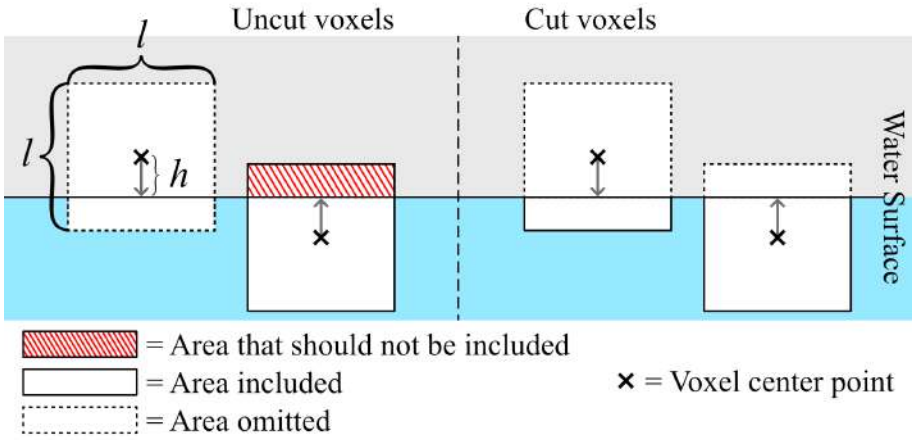



Figure 15: *An illustration of the problem we faced when using voxels, and the result of our improvement by voxel cutting.*

### 4.2.6. Hydrodynamics - Kerner Model

This approach is loosely physically based and incorporates a few key forces that attempt to account for complex hydrodynamic interactions exhibited by partially submerged vessels. These interactions are viscous water resistance, pressure drag and "slamming force". Due to time constraints, only the viscous water resistance and pressure drag have been implemented.

The faces of the physics mesh are iterated over, and the viscous drag and pressure drag on each triangle is computed individually, based on its position, orientation and relative velocity. The calculation of pressure drag includes several parameters that need to be tuned to achieve the desired response. See [78] for details on the calculation of viscous resistance and pressure drag.

A benefit of this approach is that the model is vaguely based in hydrodynamic theory, though significant simplifications are made for performance reasons and complexity of implementation. One downside is that the parameters in the pressure drag equations are difficult to empirically identify, since the resulting behaviour of the vessel is a sort of numerical integration over all triangles and can not be computed and regressed over easily, which is required for system identification.

### 4.2.7. Hydrodynamics - State Space Approximation

We also implement a simplified model that uses linear, quadratic and cubic drag coefficients with sway-yaw cross-coupling. This is a simplification which ignores complex hydrodynamic interactions, but provides an intuitive model with parameters that are relatively simple to identify.

The simplified hydrodynamic drag model is as follows:

$$\begin{cases} D_U(\vec{v}) = & X_u u + X_{uu}|u|u + X_{uuu}u^3 \\ D_V(\vec{v}) = & Y_v v + Y_r r + Y_{vv}|v|v + Y_{vvv}v^3 \\ D_W(\vec{v}) = & Z_w w + Z_{ww}|w|w \\ D_P(\vec{v}) = & K_p p + K_{pp}|p|p \\ D_Q(\vec{v}) = & M_q q + M_{qq}|q|q \\ D_R(\vec{v}) = & N_v v + N_r r + N_{rr}|r|r + N_{rrr}r^3 \end{cases}$$

where $\vec{v} = [u, v, w, p, q, r]$ is the 6DOF velocity vector of the vessel in local coordinates and $D_U, D_V ...$ are the corresponding 6DOF drag forces. $X, Y, Z, K, M, N$ with subscripts denote the damping constants, following SNAME notation [111]. The non-maneuvering degrees of freedom (heave, roll and pitch) are kept especially simple to limit the number of parameters.

## 4.3. Vessel Control

We provide several ways to control a vessel. We have a manual control system with input devices and a third person camera. Also, we have the ROS connection, that can enable us to use custom control algorithms made for ROS.

### 4.3.1. Manual Control

A gamepad controller is for many a familiar and cherished input device [112]. We have therefore seen the end-user value of adding such a control scheme to our simulator. It also contributes some special features not available with mouse and keyboard, notably analog input in the format of float values in the range 0.0 to 1.0. Mainly this is received from the triggers and joysticks, the triggers as an axis, and the joysticks as a two dimensional vector [113]. We also include the possibility to use mouse and keyboard as input.

We use Unity's InputSystem to define our actions and bindings [114]. This makes managing multiple input devices simple. Our manual controller script allows for different propeller configuration based on what the engine joints are defined as. In an outboard engine setup, we can for instance rotate the bow engines in a direction opposite of the aft engines. We clamp the rotation of the engines between realistic values, like 45 or 50 degrees in both directions, with 0 degrees being parallel to the ship forward orientation.

As the propeller joints moves with the engines, we use their position as the one to apply force to, in its forward direction. Currently, we use force values found by eyeballing their realism, but this could be improved to rather calculate realistic values based on for instance engine parameters. The same process could be applied to for instance the rotation of the propeller itself, with some realistic rotation value ramps perhaps.

### 4.3.2. Third person camera

We implement a standard version of a third person camera. It features a camera pivot GameObject which represent the point we rotate the camera around. The camera itself

is a child of this pivot, and is offset by a certain distance, using its transform component.

We have a script attached to the camera pivot that listens for input from the mouse or joystick, and then rotate the pivot dependent on the value of these inputs. An extra improvement we have included in our script is the ability for the camera to detect collisions, and then move to the surface it collides with. This avoids having the camera go through different obstacles, and rather zooming in on the pivot.

## 4.4. Sensors & Actuators

In this section, the implementation of select sensors, actuators and complementary algorithms are presented.

### 4.4.1. LiDAR

For the LiDAR sensor, the Unity `Raycast` API [115] is used to obtain $[x, y, z]$ point measurements of the surrounding environment in the sensor reference frame. Given a specified horizontal and vertical FOV (field of view) and beam count, a list of scan vectors are generated.

The Raycast method returns the distance to the first collision in the direction of the scan vector, which requires information about the current physics state. As for most game engines, the physics are performed on the CPU, which causes this approach to scale poorly as the number of scan vectors becomes large. To mitigate this, the process is parallelized using the Unity Jobs system [28]. Still, the lack of scalability in this simple CPU-based implementation remains a problem.

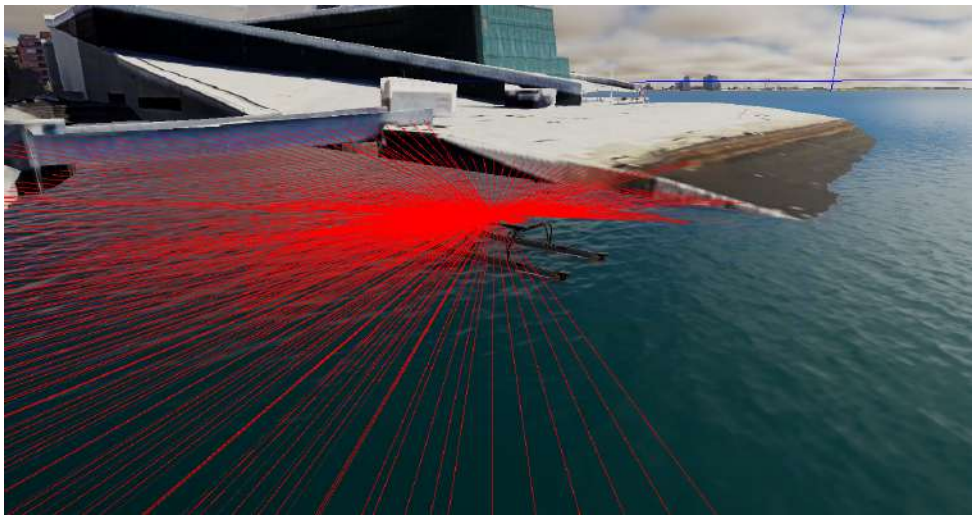


Figure 16: *Visualization of a 3D LiDAR scan*

A potential improvement would be to access the depth information from the GPU and process these in a shader to generate the point cloud.

After performing the raycasts, the resulting point cloud is processed such that it can be published in proper `sensor_msgs/PointCloud2` format. The point cloud can be visualized in the Unity Editor or in RViz2.

### 4.4.2. RGB Camera

To simulate RGB camera sensors, we use Unity's built-in camera system [116]. Unity allows multiple cameras with different parameters to be placed around the scene. However, having multiple cameras rendering the scene at the same time is computationally very expensive, as it requires the scene to be rendered multiple times.

Since the actual camera image color data lives on the GPU, it cannot be directly referenced in a C# script. There are several methods to fetch the camera texture data and copy it onto CPU memory. The two main methods are the synchronous, blocking `ReadPixels()` method and the asynchronous, non-blocking `AsyncGPUReadback()` method. The async readback method introduces some latency in the fetched data, but was chosen over the `ReadPixels()` approach for performance reasons.

The color buffer of the camera's render texture is copied to CPU memory, and the data is serialized into a byte array which is published as a `sensor_images/Image` message.

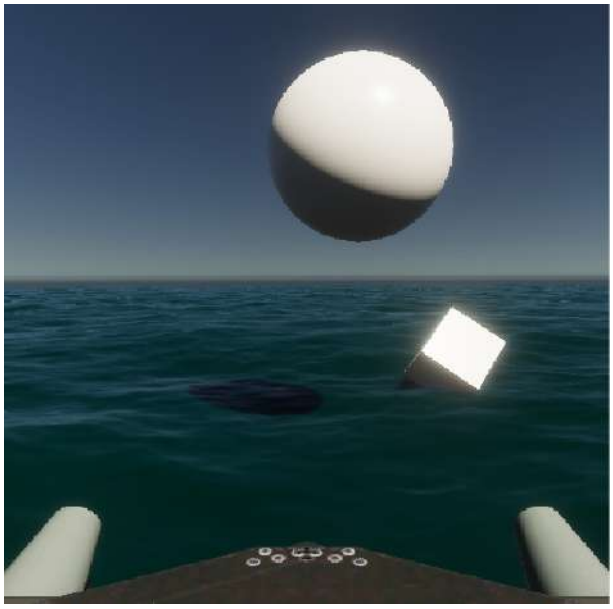


Figure 17: *RViz2 visualization of the RGB camera image stream*

### 4.4.3. Depth Camera

We include a depth camera game component that may be attached to the target camera to extract and publish depth maps from the camera view. During the rendering process, Unity writes data to depth buffers to correctly render objects to the screen from a particular camera/viewpoint, through a process known as a "depth test" [117]. This means that the rendering pipeline essentially has a built-in virtual depth sensor whose data is stored on the GPU.

Since `AsyncGPUReadback()` reads from the color buffer by default and not the depth buffer, we keep a separate render texture whose color buffer is used to store the depth data from the camera viewpoint. We refer to this render texture as the depth texture. To render the depth data into the color buffer of our depth texture, we use a custom pass volume and script [118]. The script uses the built-in `RenderDepthFromCamera()`

method, which renders the eye space depth of objects from the view point of a camera into the color buffer of the depth texture [119]. The color buffer of the depth texture is encoded as a single-channel 32-bit float texture, as it only needs to store a single depth value for each pixel. The result can be visualized as a grayscale image (Figure 18), where the intensity of each pixel represents the depth of the object at that pixel location.

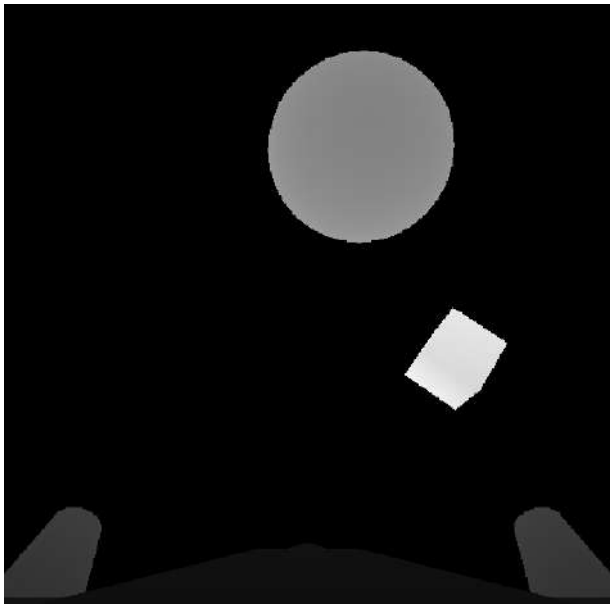


Figure 18: *RViz2 visualization of the published depth maps*

Since the depth information now exists in the color buffer of the depth render texture, we again use the `AsyncGPUReadback()` to fetch, process and publish this data in a script. The depth images are published as `sensor_msgs/Image` messages, following the standard specified in REP 118 [120].

The depth images generated using this method are *ground truth* or *rendered* depth, as it is directly computed based on objects in the scene, which are known to the Unity engine.Consequently, the resulting depth images are free of artifacts and noise that typically arise from real depth estimation techniques such as stereo vision. Instead of reading back the depth generated by the game engine directly, one could manually perform depth estimation by simulating a stereo camera system for more realistic results. This would however require additional scene rendering, compromising the goal of real-time performance. It would also require implementing a stereo depth algorithm using C# scripts and compute shaders, which is outside the scope of this thesis.

### 4.4.4. Camera Image Processing

The RGB and depth images from the camera and custom pass are raw, mostly unprocessed data directly obtained by the Unity rendering pipeline. For more realistic sensor simulation, we perform image processing on this data to simulate the characteristics of real cameras. The processing performed on the raw RGB and depth images amount to radial-tangential lens distortion and random noise. The algorithms used are imple-

mented as Unity shaders which run on the GPU, minimizing computational overhead as each pixel is processed in parallel.

To simulate the effect of lens distortion we adopt Brown's camera distortion model [121] in UV coordinates (normalized texture coordinates):

$$\begin{aligned} u_{rect} &= u_d + (u_d - u_c)(K_1 r^2 + K_2 r^4 + \dots) + \\ &\quad \big(P_1(r^2 + 2(u_d - u_c)^2) + 2P_2(u_d - u_d)(v_d - v_c)\big)(1 + P_3 r^2 + P_4 r^4 + \dots) \\ v_{rect} &= v_d + (v_d - v_c)(K_1 r^2 + K_2 r^4 + \dots) + \\ &\quad \big(2P_1(u_d - u_c)(v_d - v_c) + P_2(r^2 + 2(v_d - v_c)^2)\big)(1 + P_3 r^2 + P_4 r^4 + \dots) \end{aligned}$$

where $(u_d, v_d)$ and $(u_{rect}, v_{rect})$ are the distorted and UV coordinates, $(u_c, u_v)$ is the distortion center and $r = ||(u_d, v_d) - (u_c, v_c)||_2$ is the Euclidean distance from the distortion center to the distorted UV coordinate. $K_i$ and $P_i$ are the radial and tangential distortion coefficients.

To limit computational processing overhead and to conform to `sensor_msgs/CameraInfo`, we truncate the model to three radial coefficients and two tangential coefficients, i.e $K_j = 0$ for $j > 3$ and $T_k = 0$ for $k > 2$. We also assume that the distortion center is in the center of the image ($u_c = v_c = \frac{1}{2}$). The truncated distortion model then becomes:

$$u_{rect} = u_d + (u_d - 1/2)(K_1 r^2 + K_2 r^4) + \big[P_1(r^2 + 2(u_d - 1/2)^2) + 2P_2(u_d - 1/2)(v_d - 1/2)\big]$$

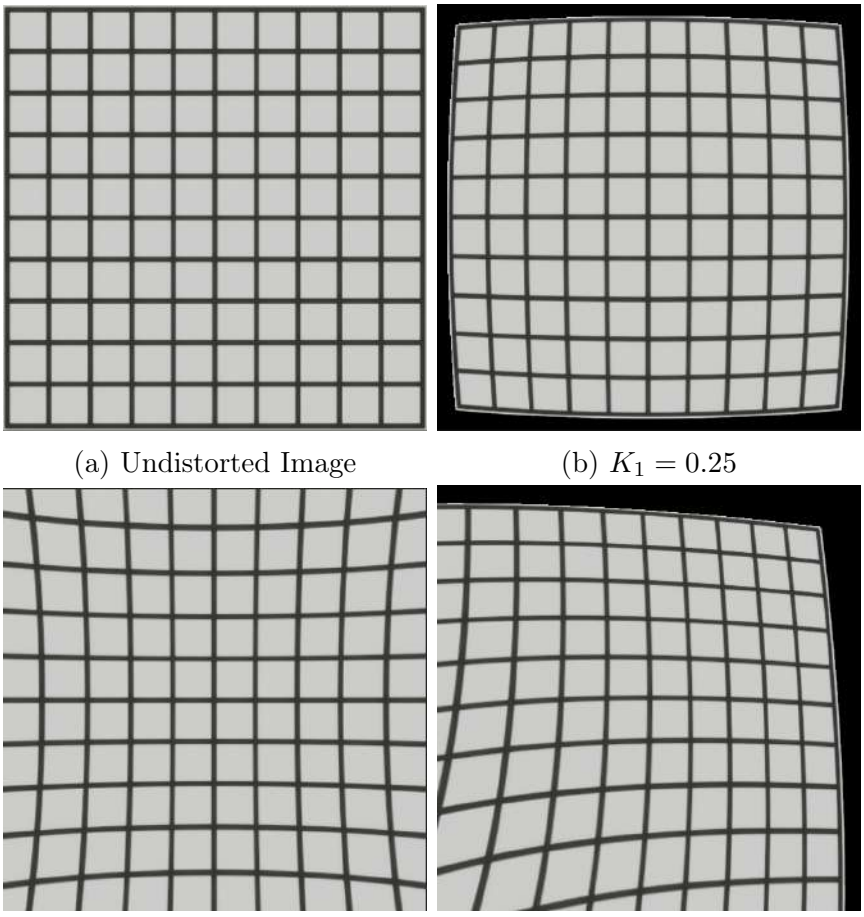
$$v_{rect} = v_d + (v_d - 1/2)(K_1 r^2 + K_2 r^4) + \big[2P_1(u_d - 1/2)(v_d - 1/2) + P_2(r^2 + 2(v_d - 1/2)^2\big]$$

This model is implemented as a Unity fragment kernel, which is a function that operates on each fragment (pixel) of the input texture. The output of the shader is another texture showing the result of the distortion. In HLSL/Shaderlab shader code:

```
// runs for each fragment/pixel in input texture
fixed4 frag (fragment i) : SV_Target
{
    // get uv-coordinate of (distorted) output fragment
    const float2 uv_d = float2(i.uv[0], i.uv[1]);

    // get uv-coordinate of corresponding (undistorted) input fragment
    const float2 uv_rect = getUndistorted(uv_d, _K1, _K2, _K3, _T1, _T2);
    // check that undistorted fragment is not outside input texture bounds
    if (outOfBounds(uv_rect)) return 0.0;
    // sample the undistorted input texture at undistorted uv coordinate
    float4 color = tex2D(_MainTex, uv_rect);
    return color;
}
```

The parameters _K1, _K2, _K3, _T1 and _T2 correspond to the radial and tangential coefficients in Brown's model. These parameters are exposed to the CPU and can be tuned in a user interface to achieve the desired distortion. Figure 19 shows the result of the distortion shader for four different sets of distortion coefficients.



(a) Undistorted Image (b) $K_1 = 0.25$

(c) $K_1 = -0.25$ (d) $T_1 = 0.1$, $T_2 = 0.1$

Figure 19: *Distortion shader output for different distortion parameters. Barrel distortion and pincushion distortion are obtained by setting positive and negative $K$ coefficients, respectively. $T_1$ and $T_2$ affect the tangential distortion in the horizontal and vertical directions, respectively.*

To account for imperfections in the data captured by physical cameras, we implement a multiplicative noise shader which adds noise to arbitrary render textures. For each fragment in the input texture, we generate a noise factor $w$ by which the pixel value is multiplied. For a given fragment $i$ the flat noise factor $w_i$ is generated by a scaled and clamped Gaussian noise model:

$$w_i = \text{Clamp}(z_i, L, U)$$

$$z_i = k\left[1 + N(\mu = 0, \sigma^2 = 1)\right]$$

Where $N(\mu, \sigma)$ represents a Gaussian distribution with mean $\mu$ and variance $\sigma^2$, $k$ is a tuning parameter and $(L, U)$ are the clamping bounds.

The sampling of the Gaussian distribution is implemented on the GPU via pseudo-random number generation. First, a pseudo-random number is generated according to a uniform distribution [122]. The sample is transformed to a Gaussian distribution using the Box-Muller transform [123]. The seed for each fragment is generated by its UV-coordinate and a time-dependent term, to induce spatial and temporal variation.

This technique is independently applied to the camera RGB and depth outputs. Intuitively, the RGB noise represents inaccuracies in color capture, and the depth noise represents inaccuracy in depth estimation.

### 4.4.5. Camera Info

The `sensor_msgs/CameraInfo` message contains important camera metadata, such as the camera projection matrix and distortion model. For real cameras, this information is usually obtained from the sensor manufacturer or through camera calibration techniques such as Zhang's method [124].

Since our camera is virtual and defined by the parameters specified in the Unity Editor, these parameters can be directly obtained through simple calculations. We use `CameraInfoGenerator` class to generate these `sensor_msgs/CameraInfo` messages, which is included in Unity's robotics library.

The projection matrix stored in the camera info message encodes the relationship between 2D pixels and 3D points. With this data provided, the depth and color data can be visualized as a 3D colored pointcloud. RViz2 has a built in visualizer which performs the back-projection and color-depth fusion automatically, provided an color image topic, a depth map topic and a camera info topic. Figure 20 demonstrates that the color, depth and calibration data are being published and handled correctly.

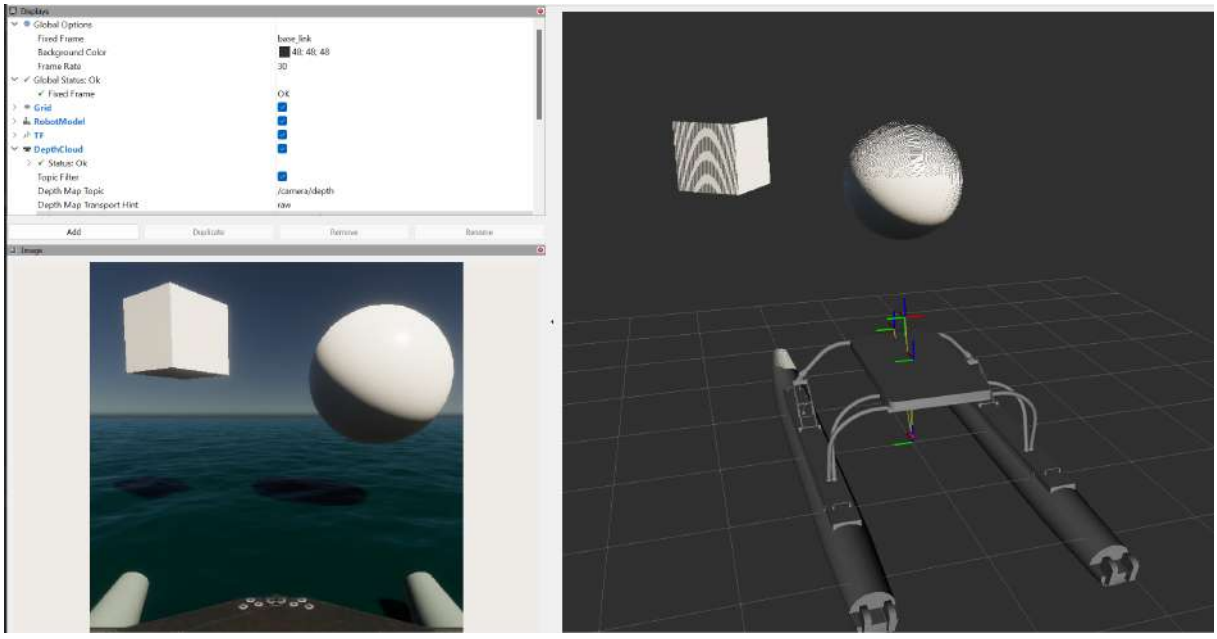



Figure 20: *RViz2 visualization of colored depth cloud.*

### 4.4.6. Navigation sensors

For navigation, control and autonomous behaviour it is necessary to have sensors that measure positional and inertial data. To support this, we include simple virtual sensors that captures and publishes this data. These components are labelled as "odometer" and "IMU" (Inertial Measurement Unit). We do not model the underlying mechanisms of real odometers or IMUs, but simply get the required state information directly from

the Unity game engine. This includes position and orientation, velocities and accelerations in six degrees of freedom. The acceleration is not directly provided, and is computed by finite differences [125]:

$$\vec{a_k} = \frac{\vec{v_k} - \vec{v}_{k-1}}{\Delta t}$$

where $\vec{a}_k$ is computed by taking the difference between the current velocity $\vec{v}_k$ and the velocity at the previous time step $\vec{v}_{k-1}$ and dividing by the time step $\delta t$. Gaussian white noise is added to the virtual measurements. with customizable parameters.

### 4.4.7. Propulsion System

The design of the propulsion system includes modular thrusters that can be independently positioned on the vessel. The thruster dynamics are modeled using two first-order linear ordinary differential equations, introducing smoothness and a lagging response to the commanded thruster effort. Mathematically, the equations governing the thruster dynamics are:

$$\dot{T} = K_T(T_r \cdot T_{max} - T),$$
$$\dot{\theta} = K_\theta(\theta_r \cdot \theta_{max} - \theta),$$

where $T$ and $\theta$ are the actual thrust magnitude and direction, $T_r$ and $\theta_r$ are reference signals on $[0, 1] \times [0, 1]$, and $\theta_{max}$ and $T_{max}$ are the user-specified force and angle limits. The reference signal channels are exposed as ROS 2 topics, allowing the user to transmit control signals during simulation.

### 4.4.8. Automatic Velocity Control

Automatic control of the surface vehicle is typically not included as part of the simulator. Instead, it is left to the discretion of the end user to implement, as controllers are highly specific to the target vehicle and its requirements. However, basic automatic velocity control is required to test certain use cases presented in Section 5., such as autonomous navigation. Therefore, a simple controller has been implemented as a ROS 2 package.

The controller consists of two independent Proportional-Integral-Derivative (PID) controllers [126], taking in linear and angular velocity commands. The output of these controllers is a force and torque in the surge and yaw direction, respectively. The controller is implemented as a ROS 2 timer callback, which is called at a constant rate. Since the controller is implemented in ROS 2, it is completely independent from the Unity simulation, only communicating via the TCP connector.

### 4.4.9. Thrust Allocation

Since the velocity controller outputs a generalized force vector, a thrust allocation scheme is required. The task of the thrust allocator is to generate individual thruster commands, such that the total propulsion force on the vessel tracks the commanded force. Figure 21 shows the structure of the velocity control and thrust allocation scheme. `cmd_vel` takes in velocity commands and produces generalized force commands. `cmd_force` then generates individual thruster forces and angles, which are sent to the simulator.

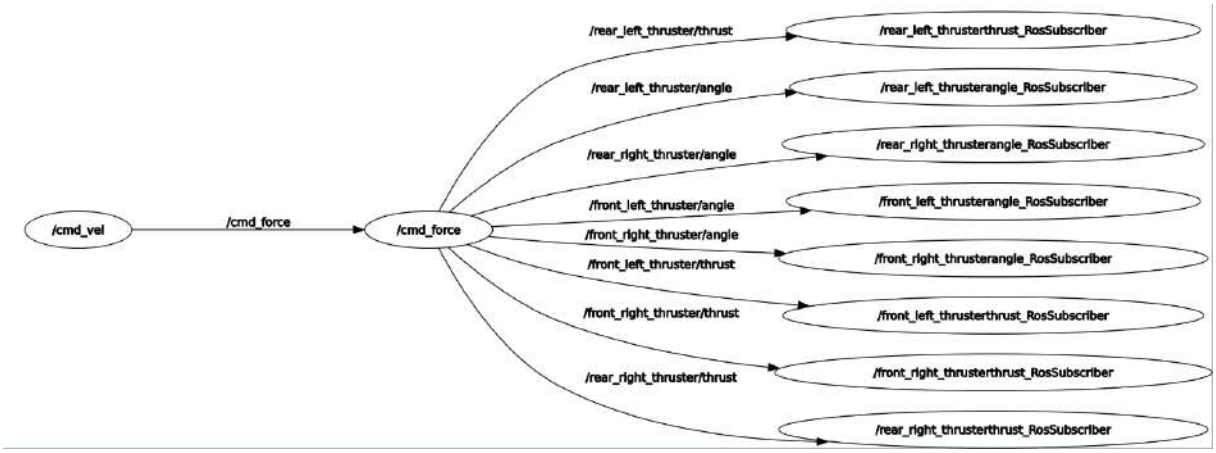



Figure 21: *Graph showing the structure of the simple velocity control and thrust allocation implementation. The graph was generated by rqt_graph[54]*

For our thrust allocation implementation, the vehicle has four rotatable thrusters at known positions. We also assume that the thrusters are bi-directional, and can be rotated at least 180 degrees.

In summary, the thrust allocation node solves the following optimization problem:

$$\min_{\vec{u}} \vec{u}^T \vec{u}$$

$$\text{subject to: } \vec{\tau} = B\vec{u}$$

Where $\vec{u}$ represents the individual thruster forces in Cartesian coordinates, $\vec{\tau}$ is the generalized force input vector, and $B$ is the thrust matrix, encoding the positions of the thrusters relative to the vessel center of gravity. Intuitively, we find the smallest thrust vector (in the $L^2$ sense) that produces the commanded force $\tau$. This optimization problem is solved using the Moore-Penrose generalized inverse [127, 128]. Finally, we compute force-angle pairs which are sent as commands to the individual thrusters.

# 5. Results

This section outlines the results from various tests designed to evaluate the performance, accuracy and usability of our simulation platform for unmanned surface vehicles. Detailed here are findings from buoyancy calculations, mesh and voxel model validations, sensor simulation effectiveness, and autonomous navigation capabilities.

## 5.1. Calculation of submerged volume

To evaluate the accuracy of the buoyancy algorithms, we compare the computed displaced volume for a shape for which an analytical solution is known. For this, we chose a sphere with a radius of R = 1. The analytical solution for the water displacement of this shape is calculated using the formula for the volume of a spherical cap:

$$V = \frac{\pi d^2}{3}(3R - d)$$

where $d$ is the vertical distance from the bottom of the sphere to the water surface. The triangulated spheres were generated by icosphere subdivision in Blender for different levels of detail [46]. A custom script was made to log the calculated displaced volume as the spheres were lowered into a flat water surface. The calculations were performed using the method presented in 4.2.4.

The data gathered from this experiment is shown in Figure 22 and table 1. For the sphere approximated using 80 triangles, the error reaches a peak of $e_{coarse} = 0.0662m^3$, with a relative error $\delta_{coarse} = 12.655\%$. This error is reduced to $e_{detailed} = 0.0011\%$ with a relative error of $\delta_{cdetailed} = 0.22\%$, for the sphere with 5120 triangle faces.

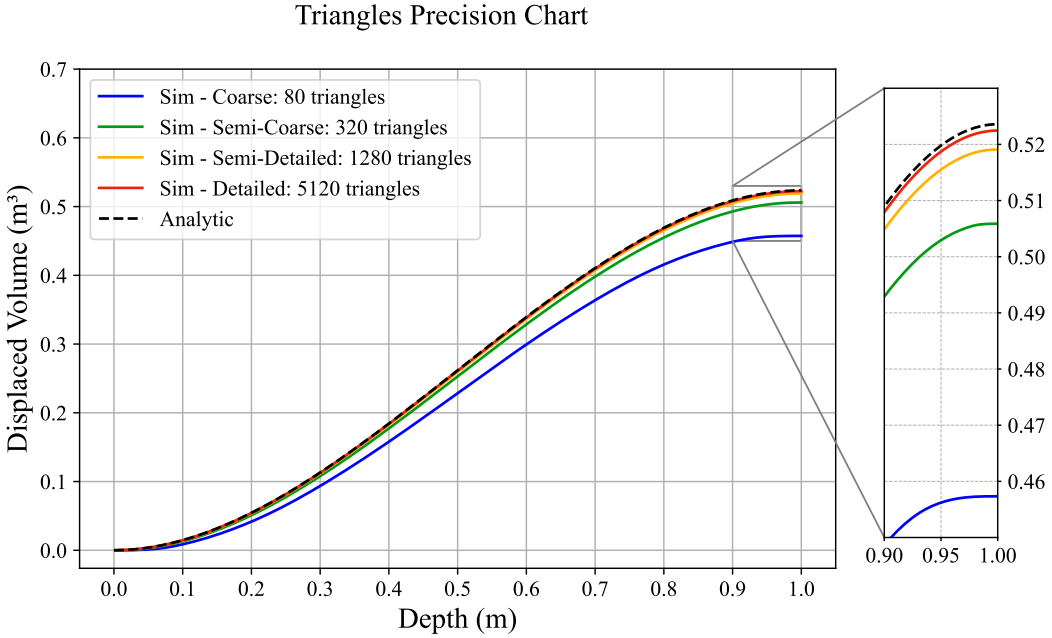



Figure 22: *The calculated displaced volume for different levels of mesh detail compared to the analytical solution.*

| Number of Triangles | Max Error | Max Error % |
|---|---|---|
| 80 | 0.0663 $m^3$ | 12.65 |
| 320 | 0.0177 $m^3$ | 3.38 |
| 1280 | 0.0045 $m^3$ | 0.86 |
| 5120 | 0.0011 $m^3$ | 0.22 |

Table 1: Error Analysis

The same experiment was performed using the voxel method, both with and without voxel cutting. The results showed that there was major increase in accuracy to be had from halving the size of the voxels. However, with each halving of the diameter, the number of possible voxels increase by a factor of eight. The actual amount of voxels inside the mesh does not increase as much, because the precision of the discrimination also improves. See Figure 23.

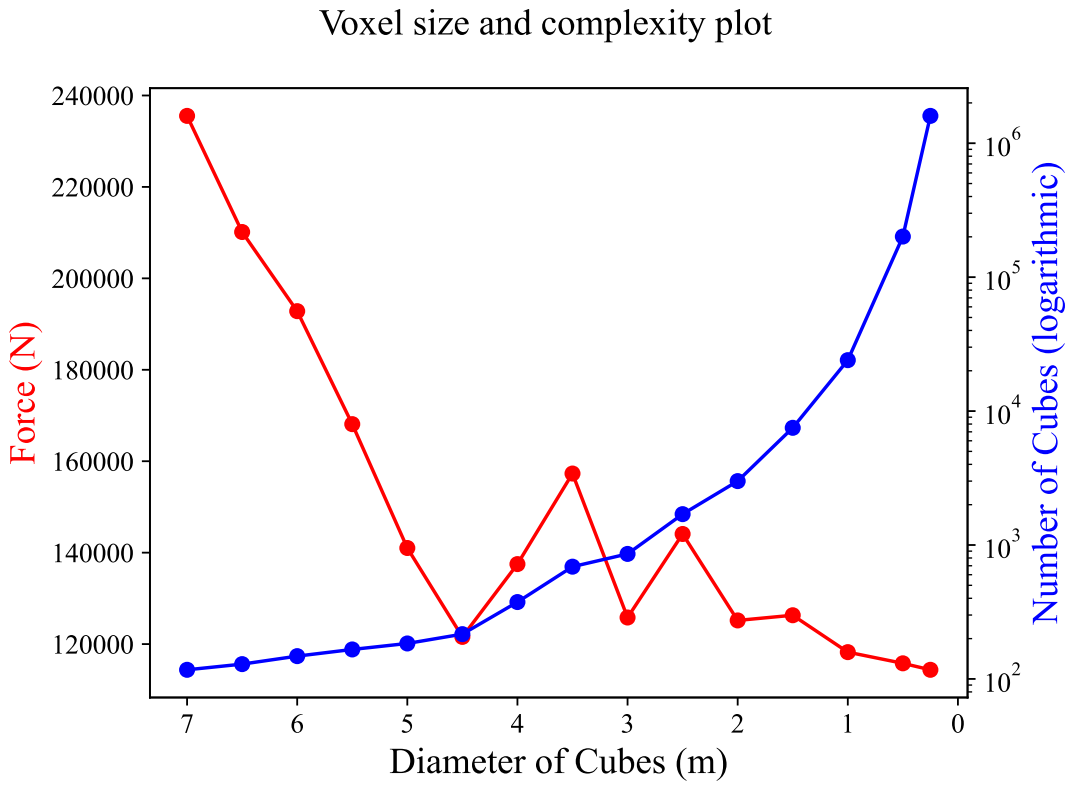



Figure 23: *Graph plotting the force generated on the left vertical axis, and the number of cubes on the right vertical (logarithmic), based on diameter of each cube on the horizontal axis. The precision increases as the difference in force from one step to the other is reduced. We approach an equilibrium when the amount of force converges.*

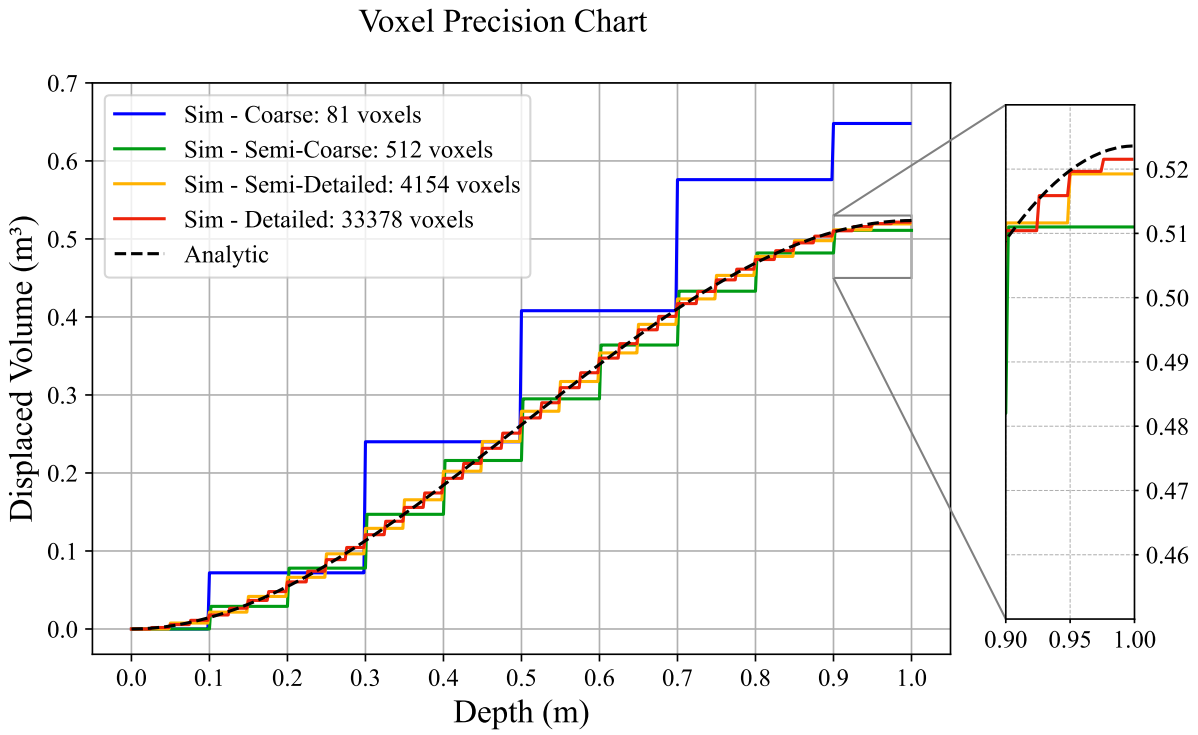



Figure 24: *Plotted lines show the volume of different resolution voxels in relation to the analytical solution. Notice the step discontinuity.*

As mentioned in chapter 4.2.3., the waterline of the partially submerged vessel is approximated by cutting partially submerged mesh triangles into smaller, fully submerged triangles. A result of this procedure is a smooth relationship between the depth of the object and the approximated water displacement. A simpler approach would be to discard triangles whose vertices are not all submerged. However, this results in discrete jumps in the calculated displaced volume. For meshes with a low triangle count, this would in turn result in significant discontinuities in the applied buoyancy force, which is undesirable.

This effect is shown in Figure 25. When a triangle becomes fully submerged, there is a sudden jump in the computed volume displacement. The positive discontinuities in the graph are due to downward-facing triangles suddenly becoming fully submerged. Similarly, negative discontinuities occur as upward-facing triangles become submerged.

We implement a similar cutting concept on our voxels. The effect was even more pronounced here, completely removing the jump discontinuity seen in Figure 24. This results in increased precision between the previous discontinuities, giving a piecewise linear increase in volume, as seen in Figure 26.

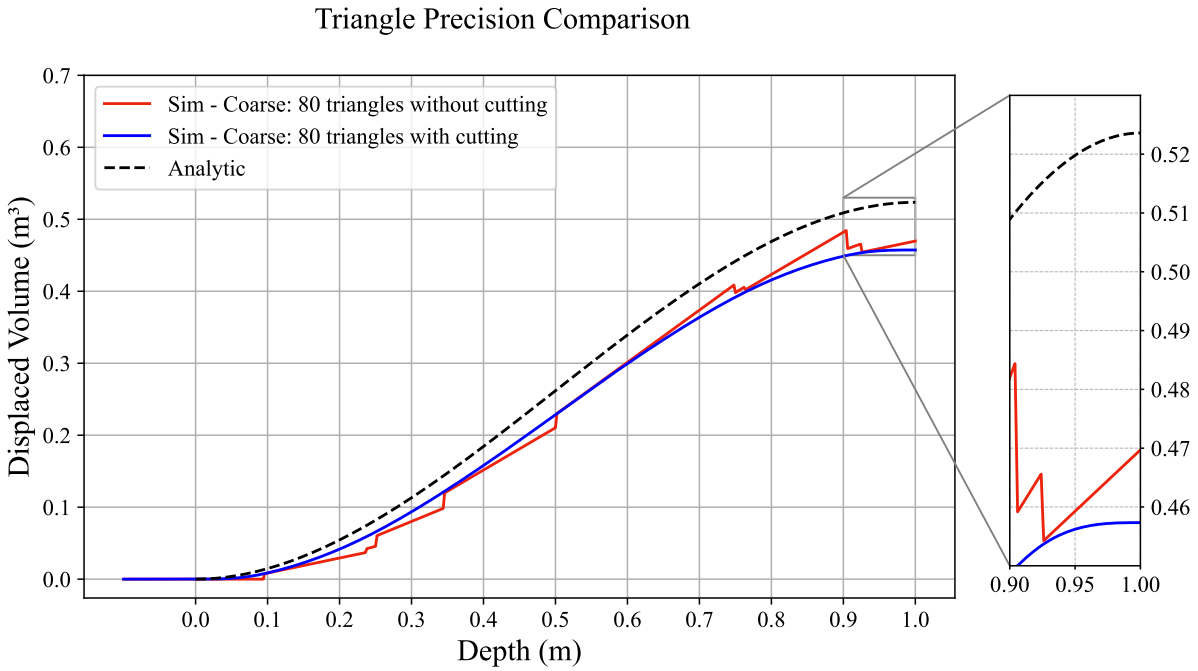



Figure 25: *The result of not performing the triangle cutting/refinement algorithm on the partially submerged triangles. This results in discontinuities when triangles go from a "dry" state to a "fully submerged" state or vice versa.*

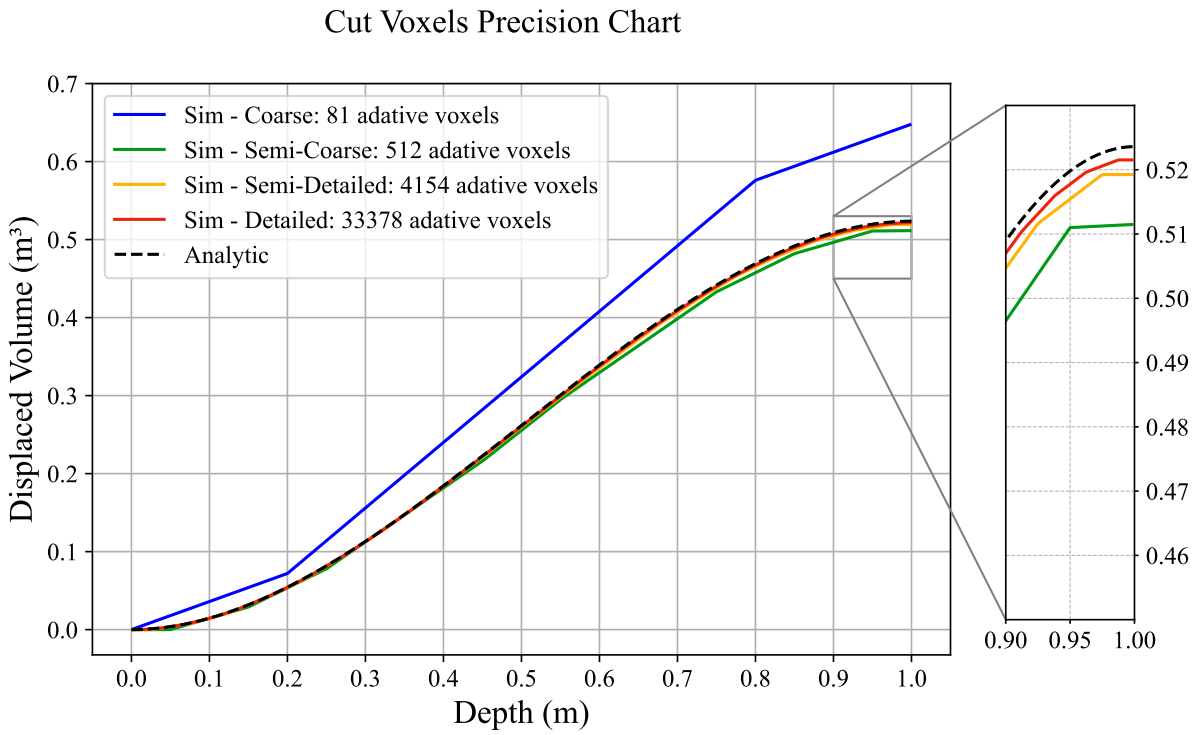



Figure 26: *Here are the same voxels as used in Figure 24, but with the voxel cutting algorithm applied. The discontinuity is removed.*

## 5.2. Performance

Preliminary measurements were performed to evaluate the processing demand of the simulator. The figures presented in this section were generated during simulation in the Unity Editor, on a desktop computer with an AMD Ryzen 5 5600x processor, an Nvidia RTX 3060 GPU and 16 gigabytes of DDR4 RAM. Figure 28 shows the stability and performance of the simulation under different loads, and figure 29 compares the performance of buoyancy calculation variants. Figure 27 shows how the performance of the LiDAR simulation scales as the number of beams increases.

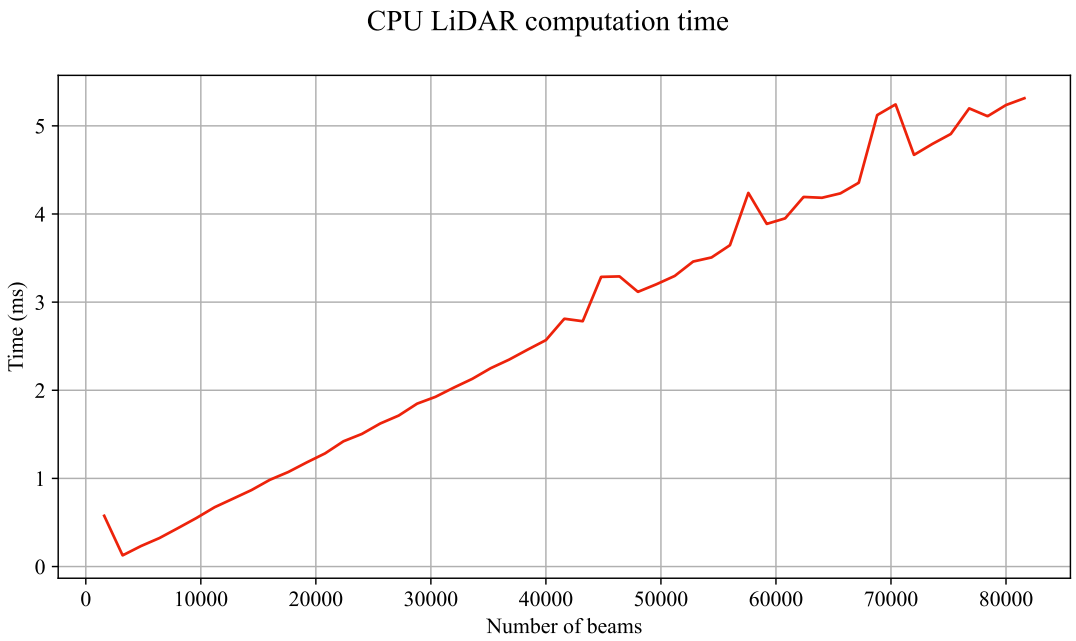



Figure 27: *The performance of the raycast-based LiDAR simulation, running on the CPU. Naturally, the computation time appears to scale linearly with the number of beams/points.*

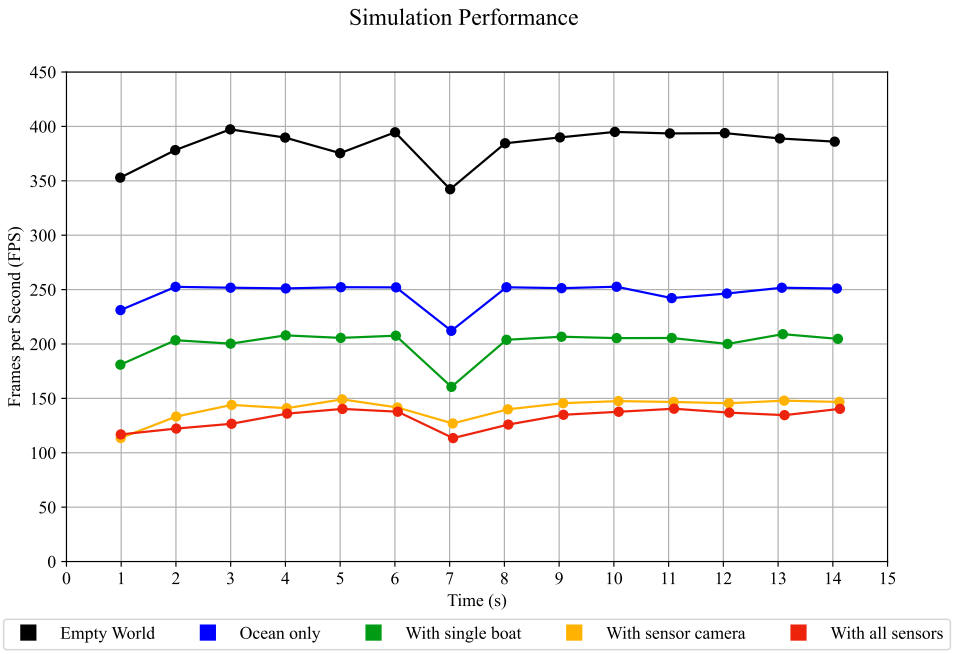



Figure 28: *The graph shows how the performance of the program is affected as game objects and features are added to the world.*

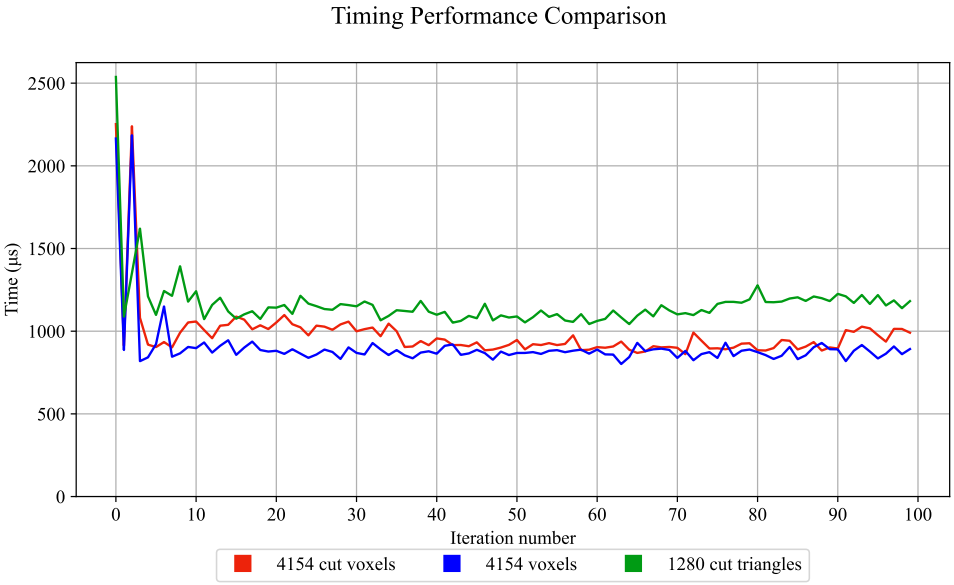



Figure 29: *Here is shown on the vertical axis the time in microseconds taken for one (1) iteration of the FixedUpdate function of our different buoyancy scripts. This is the function in which the main processing is done. Plotted against multiple iterations of the function on the horizontal axis. The choice of resolution for comparable voxels and triangles have been done studying the graphs of Figures 22 and 26 visually.*

## 5.3. Object Detection

The ability to use the simulated RGB camera output as input to an object detection model was performed and evaluated. The purpose of this experiment was not to evaluate the object detection model itself, but rather to validate that the images produced by the simulator are realistic enough to be used on a model trained only on real data.

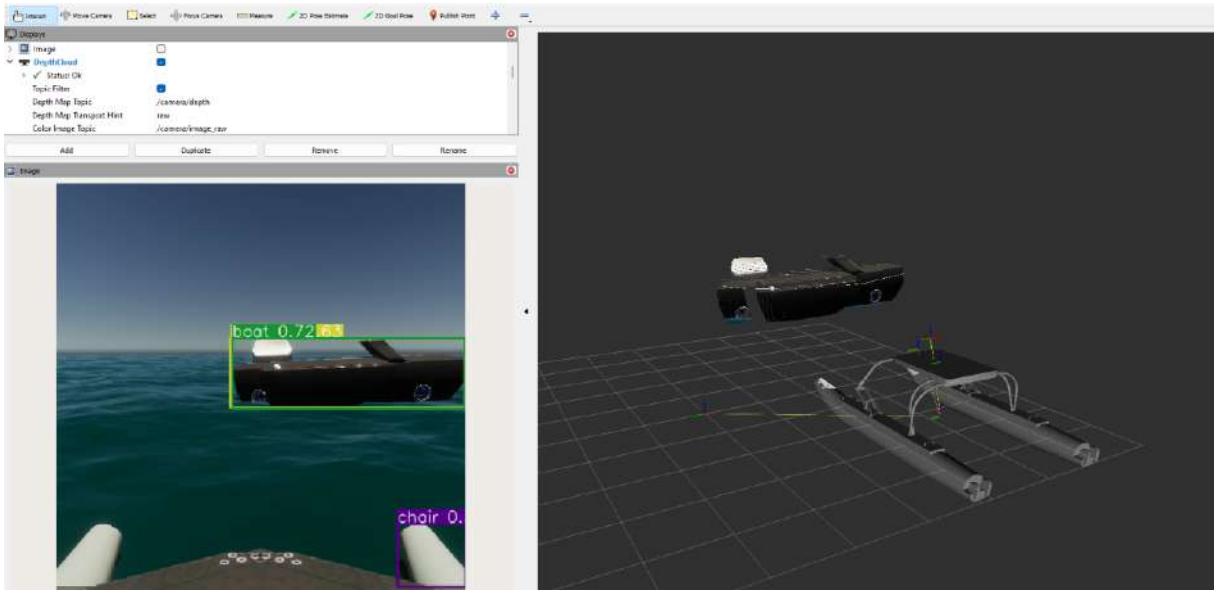



Figure 30: *Visualization of object detection and colored pointcloud in RViz2. The colored point cloud is produced using an RViz plugin which fuses an RGB stream and a depth stream using the provided camera parameters. WAMV and Navier boats pictured, credited in chapter 8.*

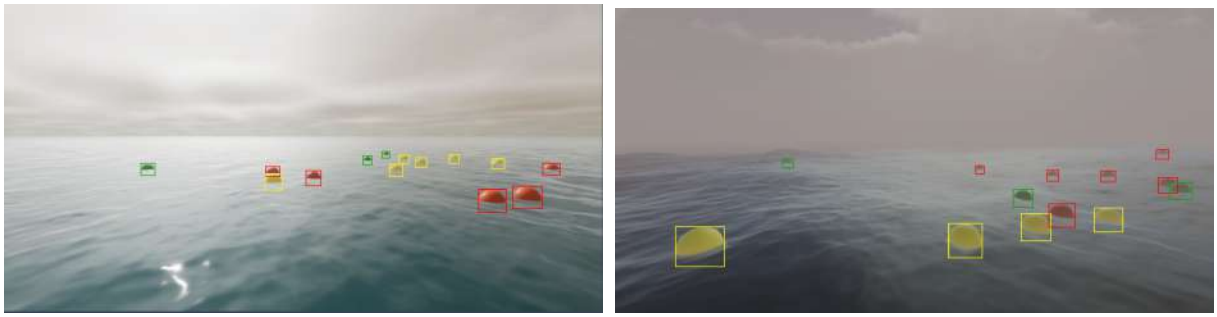
The object detection was implemented as a ROS 2 node. The node subscribes to a `sensor_images/Image` topic, performs inference on the image and publishes a plotted result image, showing the detected object classes. For the object detection itself, we used the YOLOv8 model provided by Ultralytics [129]. Figure 30 shows an RViz2 visualization with the result of the object detection node. The model correctly detects the Navier vessel as a boat class, with a 72% confidence level. The model also predicts false positives, but it should be noted that this is a pre-trained base model provided by Ultralytics, which may not be suited for this environment.

### 5.3.1. Synthetic Data Generation

Another experiment was conducted to demonstrate how synthetic data generation may be used to evaluate the performance of pre-trained models in different environments. In this experiment, two different object detection data sets are generated. One data set is generated in clear weather and calm seas. The other data set is generated with high levels of volumetric fog and moderate ocean waves. Using the model, inference is performed on the two data sets, and the results are compared.

The synthetic data was generated in our ASV simulation platform, using Unity's experimental perception package [130], which performs automatic labelling on select

objects in the scene. During the data set generation, the light direction of the sun and the horizontal positions of the buoys were randomized using custom scripts. Figure 31a, a sample from the clear dataset is depicted, while figure 31b shows a sample from the foggy dataset. The model used in this experiment was trained on images of real buoys, courtesy of Navier USN [131].

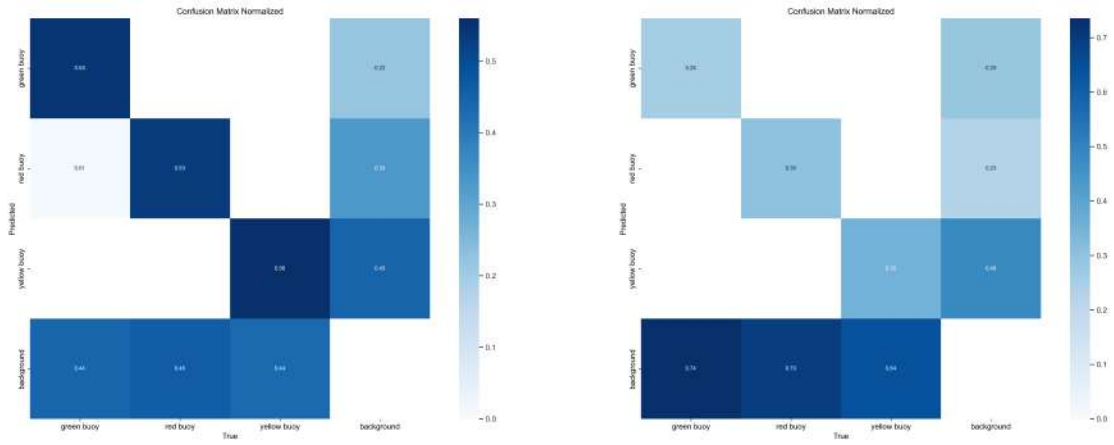


(a) *A sample from the "clear and still" data set*

(b) *A sample from the "foggy and wavy" data set*

Figure 31: *Data set samples*

The normalized confusion matrices from model validation are shown in figure 32. On most metrics, the model performs significantly worse on the foggy and wavy data set.




(a) *"Clear and still"*

(b) *"Foggy and wavy"*

Figure 32: *Normalized confusion matrices for the two validation runs, highlighting the object detection model's sensitivity to changes in the environment.*

## 5.4. Mapping and Navigation

The ability to perform mapping and navigation is a good benchmark for the usefulness of a mobile robot simulator, as it requires a high degree of simulator functionality to work correctly:

- The relevant sensors (usually LIDAR, IMU and GNSS or odometry) must all produce valid data in an accurate, synchronized manner.

- Navigation data and transforms must be correctly handled in the interface between the simulator and the development environment.
- The developer-side control system must correctly handle the control inputs generated by the navigation system.
- The simulated propulsion system has to handle the low-level control outputs correctly.

To perform SLAM, we set up a simple structure for the boat to navigate through while mapping. We use the `SLAM Toolbox` package [69] for mapping and localization and the ROS 2 Navigation Framework and System (Nav2) [132] stack for autonomous navigation. As goal pose commands are sent to the navigation stack, velocity commands are generated and passed down the control system chain, finally resulting in individual thrust and angle commands. The Nav2 stack takes the static map, the navigational parameters of the robot (e.g. turning radius, maximum acceleration) and dynamic obstacles into account as it produces the velocity commands.

### 5.4.1. 2D SLAM with SLAM Toolbox

The `slam_toolbox` package takes in the 2D laser scan messages and listens to the `odom` → `base_link` transform. The laser scan messages are generated directly by the simulated sensor and the transform is broadcast by a custom ROS node, which listens to the odometry and IMU messages from the simulator. The SLAM node then generates a map (represented as an image) and estimates the robot's position in that map by broadcasting a transform. Figure 33 shows an intermediate map generated by the mapping algorithm.

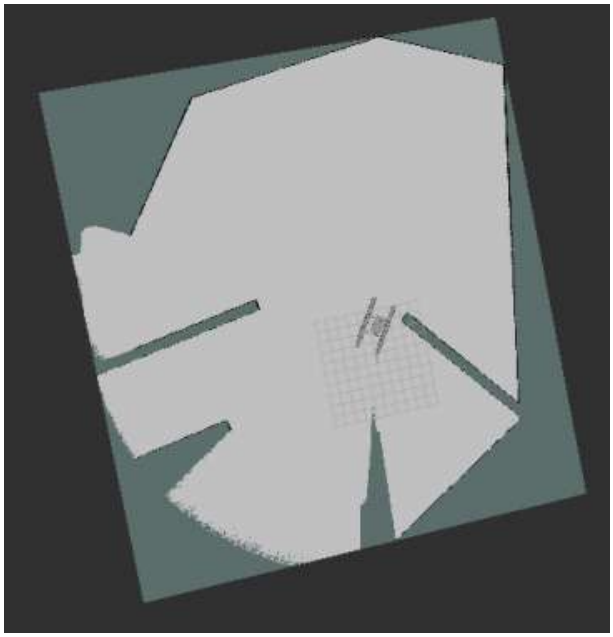
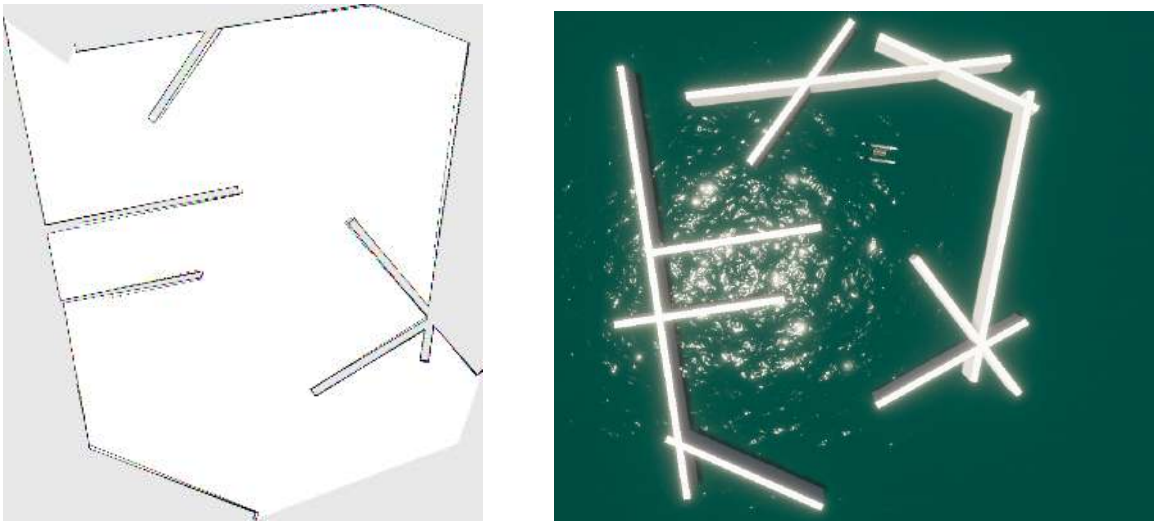


Figure 33: *Intermediate result of the mapping process, visualized in RViz2*

During mapping, the ASV was teleoperated (remote controlled) by keyboard, using the Teleop Twist Keyboard for ROS 2 [133] to discover the entire map and promote

loop closure. When the mapping is complete, the map is saved as a bitmap image file (.bmp) for later use, e.g for autonomous navigation. Figure 34a show the result of one such mapping experiment.



(a) Final bitmap representation (b) Actual map in Unity

Figure 34: *A visual comparison of the result of the mapping process and the actual map.*

### 5.4.2. 3D SLAM with RTAB-Map

3D mapping with Real-Time Appearance-Based Mapping (RTAB-Map) [134] is a SLAM algorithm that can be used to generate a colored point cloud of an environment based on various sensor data. RTAB-Map was performed using `rtabmap_ros` [134], to test and validate the RGB and depth images generated by the simulator. The package is provided with inertial and navigational data as well as RGB and depth images through the ROS 2 topics being published by the simulator.

A basic test was performed: the boat was operated via teleoperation to map the entire maze structure, revisiting previous locations to promote loop closure. The output of the RTAB node is a colored point cloud representing the 3D environment. See figure 35. The highlighted artifact on the map is due to the fact that a part of the boat is constant in the camera frame. This area can be masked to have the mapping algorithm ignore it, and the artifact would disappear.

By visually comparing the result in Figure 35 to the 2D map in Figure 34a, it is apparent that the RTAB mapping was successful. This experiment is not meant as a careful study of the accuracy of the generated maps, but is used as a stress-test for validating the consistency of the simulator and the ROS 2 compatibility.

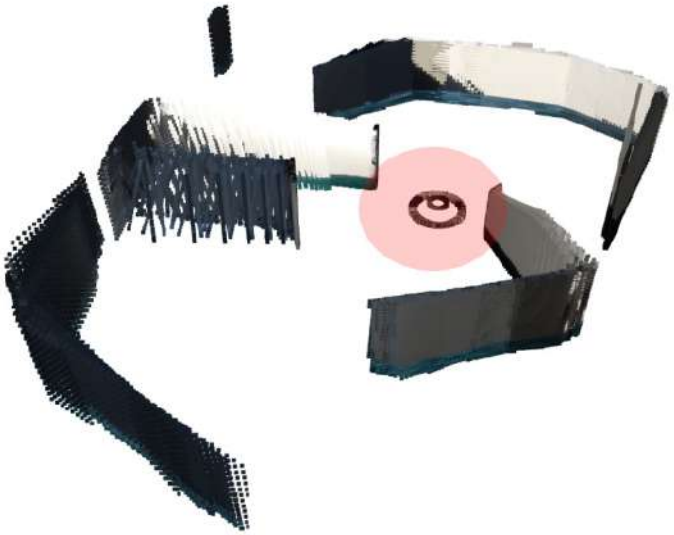


Figure 35: *Result of RTAB-Map with default parameters. The mapping artifact shown in red is due to the unfiltered boat geometry.*

Due to the nature of loop closure, the RTAB-Map algorithm is highly sensitive to changes in the environment during mapping. This is generally not an issue for indoor robots in controlled environments, but will be problematic for mapping in dynamic environments such as water bodies. Water is inherently dynamic, with its surface constantly moving due to waves, currents, and wind. This movement means that features captured by the camera at one moment generally won't be in the same position when the area is revisited, making loop closure difficult.

Two two mapping trials were performed to investigate the effect of ocean waves on loop closure. The ASV was kept at a constant position and a constant rotational velocity was applied in the yaw direction. In the first trial, local wind was applied in the water system, causing ripples. In the second trial, wind was disabled, leaving the water perfectly flat and still. Table 2 shows the results from the two trials.

| | **Still Water** | **With Waves** |
|---|---|---|
| Elapsed time (mm:ss) | 01:56 | 02:01 |
| Images sampled: | 180 | 196 |
| Loop closures detected | 35 | 5 |
| Loop closures rejected | 42 | 88 |

Table 2: *RTAB-Map results for an environment with completely still water and with active ocean waves.*

These results show a clear sensitivity for loop closure in the presence of ocean waves. Figure 36 shows an example of successful and failed loop closure detection, highlighting

how the movement of the ocean affects the RTAB-Map algorithm. The left loop hypothesis is rejected as the algorithm fails to map features found on the ocean surface to features in the previously captured frame.

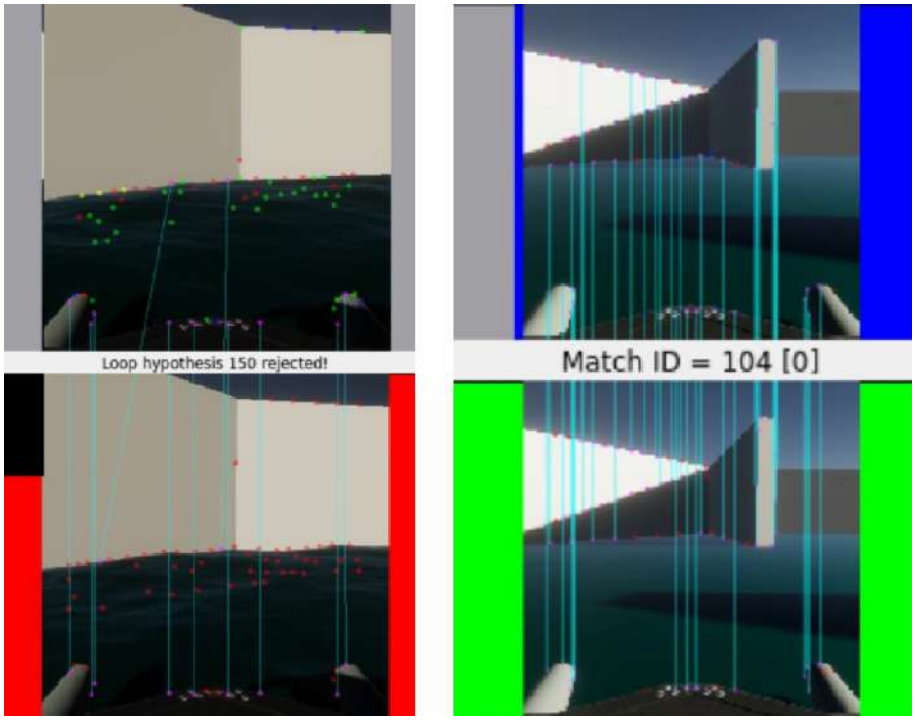



Figure 36: *Rejected and accepted loop closure. The image pair on the left shows a rejected loop hypothesis with ocean waves enabled. On the right, a loop hypothesis is accepted, with ocean waves disabled.*

### 5.4.3. Autonomous Navigation

Autonomous navigation in an unknown environment was performed to further validate the usefulness of the simulator. The same environment as the one in chapter 5.4.1. was used, and the mapping and localization was performed online, instead of pre-generating the map. This autonomous navigation task acts as a stress test for the entire simulator, as it requires correct integration of various components such as sensor simulation, control and ROS 2.

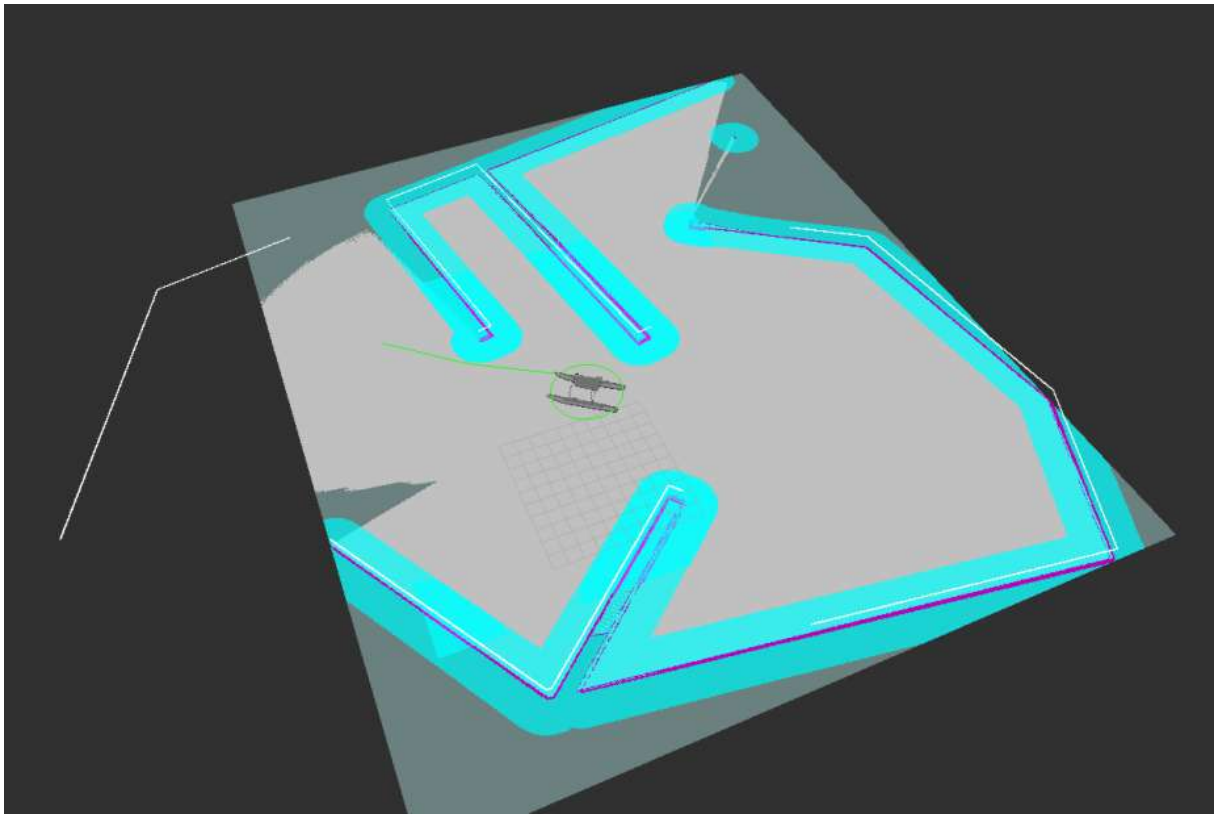


Figure 37: *An RViz2 snapshot visualizing the cost map and planned path. The costmap is represented as blue and purple areas representing the cost (risk) associated with walls and obstacles. The green line shows a planned path for an autonomous navigation goal.*

Goals are sent through RViz2 using an interactive cursor for specifying the goal position and 2D orientation of the ASV. Nav2 then generates a global and local plan for the ASV to track, as well as velocity commands leading the robot along the path. The navigation stack assumes correct handling of these velocity commands with adequate reference tracking. This is handled by the custom velocity PID controllers discussed in chapter 4.4.7.

During the experiment, the navigation stack successfully generated paths and velocity commands, which were correctly forwarded to the custom low-level controllers. At low velocities (max 0.5m/s), the ASV was able to track most paths, including paths around obstacles. Figure 38 shows a successful attempt at navigating around a corner.

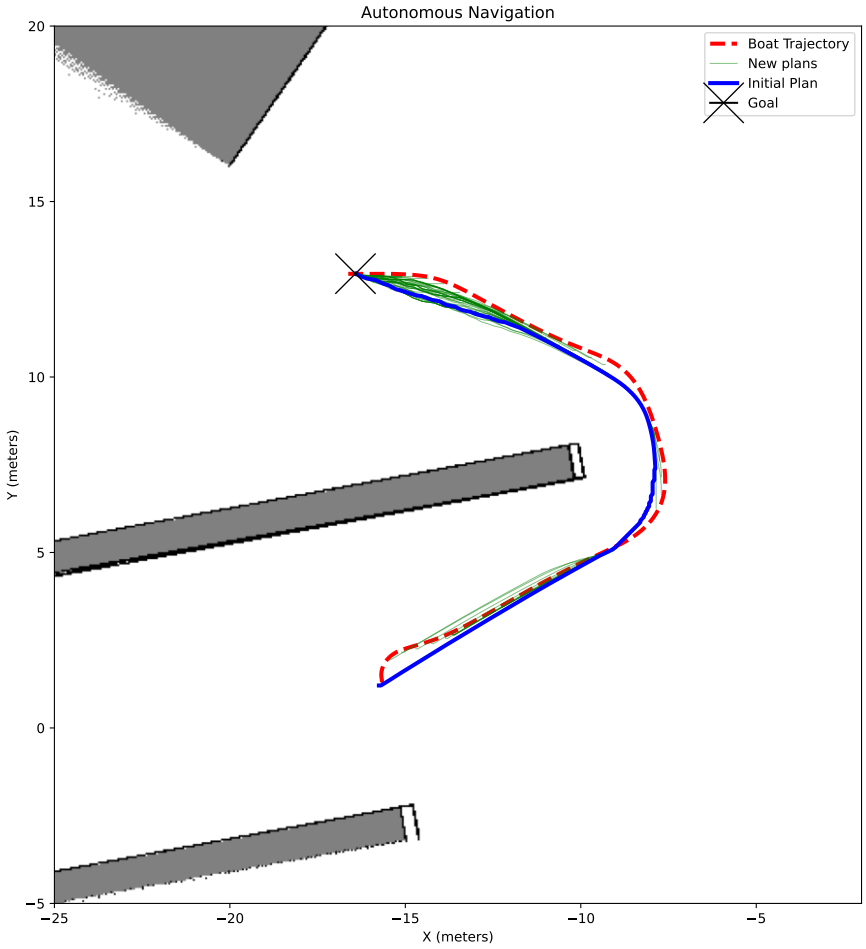



Figure 38: *Successful autonomous navigation around a corner.*

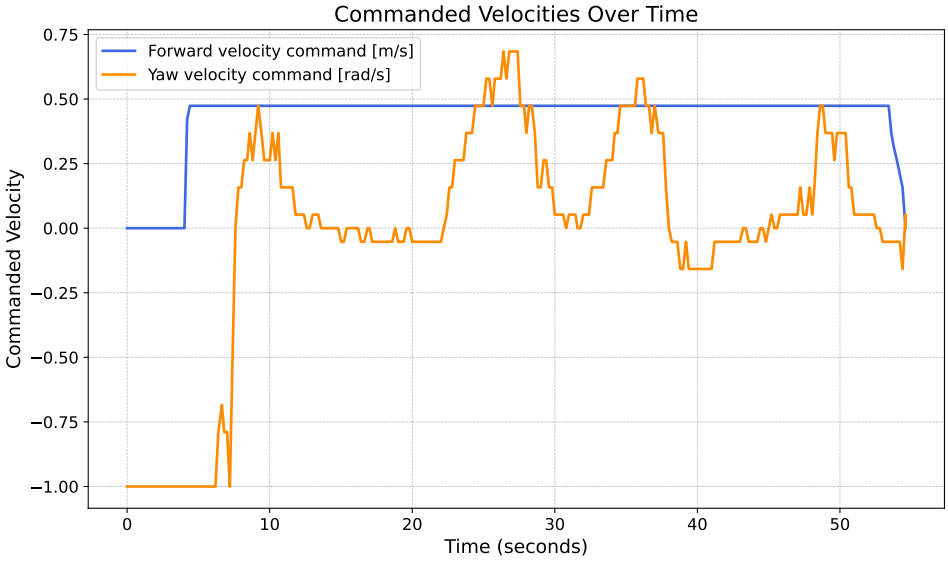



Figure 39: *Linear and angular velocity commands generated during autonomous navigation.*

At higher speeds, the ASV was able to follow simple paths, but struggled to track paths with significant curvature, and would often get stuck while navigating around corners. The commanded velocities resulted in oscillatory behaviour around the planned path. We attribute this partial failure to the lack of careful tuning of the Nav2 parameters and the PID controller gains.

Choosing the correct navigation parameters and tuning custom controllers is generally up to the user of the simulator and will depend on the specific vessel. The experiment shows that the simulator itself is mature enough to properly integrate with complex ROS2 frameworks such as Nav2.

## 5.5. World Customization

A sample scene was created to showcase the simulator's customization options, designed to meet the specific needs of the developer. Cesium was used to generate photorealistic 3D world models using Google's Photorealistic 3D Tiles [135, 136].

Figure 40: *WAM-V pictured in the sample scene. Geodata of Oslo, Norway accessed through Cesium/Google*

Custom visual effects are also added using the HDRP Volume system to produce special lighting- and post-processing effects. The scene uses volumetric clouds and a physically-based sky, which are HDRP stock assets.

Figure 41: *Sample scene with special lighting and post-processing effects, including fog, film grain, bloom and lens flare*

# 6. Discussion

In this section, we discuss the main findings of our work on the Unity-based Digital Twin & Simulation platform for unmanned surface vehicles. We examine the challenges of adapting Unity to support complex robotics simulations and how the results presented in the thesis support the goal of creating a tool for developing, testing, and validating USV algorithms.

## 6.1. Software & Performance

Unity was chosen as the simulation engine for this project, with the High Definition Rendering Pipeline. This choice allowed us to create and render high-quality materials and environments and streamlined the development process. Overall, we found Unity to be a well-suited tool for developing our simulation, as the user-friendly interface and comprehensive suite of tools enabled rapid prototyping and design iteration.

Since Unity is fundamentally a video game engine rather than a robotics simulator, it lacks built-in support for common sensors such as LiDAR and RGBD cameras, which must be implemented through custom scripts. In such cases, ensuring correctness and real-time performance requires some expertise in programming and Unity-specific development practices. For instance, implementing the RGBD camera required knowledge about Unity's rendering pipeline to efficiently extract depth images from the engine using custom passes and shaders for image post-processing.

One of the major bottlenecks of the simulation was the 3D LiDAR sensor. Since raycasts are run sequentially on the CPU, there is a linear relationship between the number of simulated beams and the computation time, as shown in Section 5.2. This highlights the need for more efficient use of resources and sophisticated algorithms to achieve real-time performance of sensors with high data throughput.

The inclusion of a main scene camera and additional sensor cameras required the scene to be rendered multiple times. This multi-rendering process substantially decreases performance, as each camera demands a separate rendering pass from the graphics backend. This performance hit cannot be avoided altogether but can be mitigated by decreasing the resolution and frame rate of the cameras. It is important to note that lowering the sensor camera frame rate might not pose a significant problem, as many robotic machine vision tasks do not require very high frame rates. Additionally, bandwidth is often limited in data transfer and computation, making lower frame rates and resolutions a practical consideration. Finally, the physics calculations running on the CPU also impact performance as more vehicles are added to the simulation.

Overall, we found the performance of the simulator to be satisfactory, though there is room for improvement. We could comfortably simulate multiple vessels with multiple cameras and sensors, simultaneously doing computationally intensive tasks such as object detection and simultaneous localization and mapping, on a single computer. Special care was taken to reduce memory usage and optimize for performance, minimizing stalls due to Unity's garbage collector. For instance, the CPU LiDAR simulation leverages the Job System to perform multi-threaded ray casting. Using GPU shaders to perform image processing and reading back textures asynchronously also significantly reduced the performance cost associated with the sensor camera simulation.

## 6.2. Physics

In general, the physics implementation produced visually believable results when applied to a variety of floating objects (Figure 42). The proposed physics system is highly modular, making it easy to test its individual components. For instance, the triangle-based hydrodynamics implementation could easily be swapped out for a state-space implementation, even during simulation. This allowed for clearer comparisons between different parts of the physics model.

Although a modular physics framework offers certain advantages, adopting a more unified approach to physical simulation may be more suitable in some cases. The end user is not guaranteed to have the background required to reason about the different modules and implementations effectively. In such cases, a well-defined, rigid physics system could free the user from having to weigh different options. As a middle ground, one could maintain the current physics system and provide sample vehicles with "tried and tested" parameters. While such a reference model may not exactly represent the vehicle that the user wishes to simulate, it allows for quick setup and offloads the parameter-tuning process.

While the buoyancy calculations are compared with analytical results in Section 5.1., no thorough analysis is done regarding the accuracy of the hydrodynamics implementation. Simulating hydrodynamics in real-time is a highly complex problem. We obtain a believable and useful vessel response by using simplified hydrodynamic models that are empirically validated or loosely physically based. Properly validating the realism of such models and our implementation of them requires physical testing and system identification which is outside of the scope of this thesis. In practice, our initial validation of these models amount to visually inspecting the resulting motion, making only qualitative evaluations of the response. This is especially true for our implementation of the Kerner-based water interaction [78]. The lack of analytical validation is a significant omission, as it compromises the validity of the physics model, and leads to an undefined "sim-to-real" gap which could affect the simulator's ability to serve as a test bed for USV control systems.

Our water interaction model only considers fluid-to-solid interactions, i.e. the water in the simulation is not affected by the movement or presence of floating objects.


Figure 42: *Simulation of different vessels: Maritime Robotics' Otter [5], the WAM-V and the USN Navier vessel.*

Ideally, an empirical analysis would be performed, comparing data gathered from simulation experiments to data from real-life experiments. However, this is a complex and resource intensive task falling outside the scope of this thesis. The main focus of our platform is not physical accuracy, but to provide a tool that can generate visually satisfying, physically plausible and *useful* simulation, in the context of USV development.

## 6.3. Applications to Robotics Development

One of the key motivations for building the USV Digital Twin & Simulation platform was to provide a way for USV robotics developers to easily prototype and test typical tasks and missions in the context of autonomous operations. The results presented in Sections 5.3. and 5.4. aim to substantiate this ability.

The results from Section 5.3. show that the simulation platform can be used to test, evaluate and validate machine learning models for object detection. We also show that using Unity as the simulation engine, with the HDRP rendering pipeline allows for quick customization of the environment, which can be used to test the robustness of pre-trained models. This was possible thanks to Unity's rich package ecosystem, which allowed us to easily set up an automated pipeline for synthetic data labelling (Section 5.3.1.).

In Section 5.4., we demonstrate the maturity of our platform by performing autonomous mapping and navigation tasks. This demonstrate the validity and correctness of our sensor implementations. It also showcases the simulator's level of integration with the ROS 2 framework.

Autonomous navigation is performed in 5.4.3. using Nav2 [132] and a rudimentary automatic velocity controller. This result further proves that the implementation of

the platform conforms to ROS 2 conventions, allowing it to easily integrate with open source robotics packages.

Finally, Section 5.5. shows how real-world geospatial data can be easily incorporated into a virtual scene. This opens the door for richer synthetic data generation and testing of autonomous navigation algorithms.

## 6.4. The voxelization

We encountered some difficulty in implementing the script for developing the voxel equivalent of our mesh. Our method as outlined in 4.2.5. uses a point cloud and ray casting to determine which points to discard from our list of voxels.

There are limitations to our method, specifically the fact that it requires a convex collider, which makes applying it to a divided hull like that of a catamaran, difficult. One way to do adapt to this use case would be to do the voxel calculation for each of the hulls at the world origin, and then move these out to their proper place at runtime.

Currently, the voxelization requires the mesh to be at the center of the world, ergo coordinates (0, 0, 0). After the voxelization process, the target object may be freely translated and rotated in world space, as the points will be snapped to this location during runtime.

The potential to improve the precision of our voxel buoyancy system was clear in its discontinuities. We made a cutting algorithm that can exclude or include more parts of the volume if the cube is semi-submerged. We are very satisfied with the results shown in Figure 26. The fact that the performance impact is almost negligible, see how in Figure 29, is another bonus of the improved voxel buoyancy model.

## 6.5. Set-up & Usability

Some models are too detailed to be rendered in real-time along with the simulation environment and other models placed in it. This problem can occur when importing models directly from the Computer Aided Design (CAD) software that the model was originally developed in. It is not uncommon for such models to contain millions of mesh triangles, which is undesirable in real-time rendering. It is also possible that the original model files uses file formats that are incompatible with Unity, such as STEP files [137]. Such cases cannot be automatically handled and the user must simplify the meshes and convert the model into a Unity-compatible format, which requires some experience with 3D modelling software.

# 7. Conclusion

This thesis presents the implementation, results, and findings of our work on a Unity-based simulation and digital twin platform for USV/ASVs. The results show the effective integration of high-definition rendering, a complete water interaction physics model, and real-time sensor data processing. These features aim to enable rapid prototyping and validation of algorithms for ASVs. The platform's effectiveness as a valuable tool for robotics developers is demonstrated by the successful integration of software packages that solve key ASV tasks, such as SLAM and autonomous navigation.

While our results show considerable promise, there are areas for improvement and expansion, particularly in enhancing realism, optimizing performance, and broadening the range of simulated interactions and sensor inputs.

Our tool can support multi-vessel simulation and customizable camera and sensor simulation with a high degree of ROS 2 interoperability. The simulator incorporates a modular water interaction model that produces visually believable results, which can be easily customized to suit the needs of different applications and vessel types. Overall, the work presented in this thesis is a successful and valuable contribution to the field of USV simulation, leveraging modern pipelines and techniques to create a user-friendly simulation environment that enables efficient development and testing of unmanned and autonomous surface vehicles.

## 7.1. Future Work

The platform presented in this thesis provides a solid foundation for digital twinning and simulation of ASVs. However, to improve the realism, performance, and completeness of the platform, we propose the following topics for future work:

**Solid-To-Fluid Interaction.** Currently, the ocean simulation is completely unaffected by floating objects. While it is challenging to implement solid-to-fluid interactions into an existing water system, it would greatly increase the realism of the simulation. The HDRP water system allows for deformation textures to offset the water height around a given location. Using compute shaders, one could calculate height displacement textures based on the state of floating objects in the world and pass these textures to the HDRP water system to create waves and other forms of displacement.

**LiDAR Realism and Performance.** The current LiDAR implementation is executed on the CPU and does not include intensity calculations or other complex interactions. For dense LiDAR simulation, the performance could be improved by implementing a highly parallelized, GPU-based method. Instead of using Unity's physics API (i.e `Physics.raycast()`), one could leverage depth-buffers to generate dense point clouds instead (as in [27]), similar to how the RGBD camera is currently simulated.

**Rendering Pipeline Agnostic.** The water interaction implementation is not agnostic to the rendering pipeline, as it assumes that the HDRP Water System API is available for height querying. However, the method used is compatible with any water system that provides a method for looking up water height at arbitrary positions in world space. In addition, the sensor camera simulation uses the custom pass feature, which is only available in HDRP, and an alternative method would have to be developed to make the solution compatible with other rendering pipelines.

**Propulsion System Realism.**A more complex propulsion system with dynamic models that can be easily tuned to fit real thruster characteristics would be a valuable addition to the simulation platform. Existing literature on propulsion models provides a rich resource for this enhancement.

**Buoyancy and Hydrodynamics Performance.** The calculation of buoyancy and hydrodynamic forces is computed on the CPU by iterating over the surface elements (or voxels) of the given mesh. Since the contribution of a single surface or volume element is found independently of other elements, this calculation is easily parallelizable. By using the technique proposed in [18], one could improve the performance of the physics implementation by moving these calculations to the GPU, using compute shaders. This adjustment would require either streaming water height data from the CPU or querying the water height directly on the GPU.

**Sensor Modeling.** The sensor models presented in this thesis could be improved by implementing more sophisticated noise models and introducing artifacts that appear in real-world data. The depth images generated by the simulated depth camera lack realistic noise and contain accurate depth values regardless of the types of objects in the scene. For stereo depth estimation, this is unrealistic, as stereo depth relies on distinct features in the image to estimate depth effectively [138]. Empirical noise models could be incorporated to help the inertial and navigational sensors produce more realistic data.

**Additional Sensors.** The platform is limited in sensor simulation support. Adding common sensors such as radar and sonar would make for a more comprehensive tool. Additional sensors, such as Doppler Velocity Loggers (DVLs), Infrared (IR) cameras, and hydrographic surveying sensors, could further enhance the development and testing capabilities of the platform.

# 8. Appendix A: Third party Credits

## 8.1. The WAM-V

From thesis proposal:

We are, for the purpose of testing and comparison, using a slightly modified version of the boat model called "WAM-V" created by Brian Bingham, Carlos Aguero, Kevin Allen, Jose Luis Rivero, Tyler Lum, Marshall Rawson, Rumman Waqar and Jonathan Wheare. It is distributed by Virtual RobotX via their GitHub repository, and covered by the terms of the Apache License, version 2 (January 2004) [139].

## 8.2. The Kenney Prototype Textures

We are in some scenes using the prototype textures distributed by Kenney through their website [140]. It is released under the Creative Commons CC0 license.

# 9. References

[1] A. Voulodimos, N. Doulamis, A. Doulamis, and E. E. Protopapadakis, "Deep learning for computer vision: A brief review," *Computational Intelligence and Neuroscience*, vol. 2018, 2018.

[2] "Hugin autonomous underwater vehicle (auv)," Kongsberg Discovery. [Online]. Available: https://www.kongsberg.com/discovery/autonomous-and-uncrewed-solutions/hugin/ (accessed May 15, 2024).

[3] "Kongsberg maritime and massterly to equip and operate two zero-emission autonomous vessels for asko," Kongsberg Maritime. [Online]. Available: https://www.kongsberg.com/maritime/about-us/news-and-media/news-archive/2020/zero-emission-autonomous-vessels/ (accessed May 15, 2024).

[4] "Maritime robotics homepage," Maritime Robotics, 2023. [Online]. Available: https://www.maritimerobotics.com (accessed Mar. 1, 2024).

[5] "The otter uncrewed surface vessel," Maritime Robotics. [Online]. Available: https://www.maritimerobotics.com/otter (accessed May 6, 2024).

[6] "The mariner uncrewed surface vessel," Maritime Robotics. [Online]. Available: https://www.maritimerobotics.com/mariner (accessed May 15, 2024).

[7] R. jian Yan, S. Pang, H. bing Sun, and Y. jie Pang, "Development and missions of unmanned surface vehicle," *Journal of Marine Science and Application*, vol. 9, no. 4, pp. 451–457, 12 2010. [Online]. Available: https://doi.org/10.1007/s11804-010-1033-2 (accessed May 17, 2024).

[8] A. Z. Akbar, C. Fatichah, and R. Dikairono, "Autonomous surface vehicle in search and rescue process of marine casualty using computer vision based victims detection," in *2022 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)*, 2022, pp. 1–6.

[9] H. Mansor, M. H. Norhisam, Z. Z. Abidin, and T. S. Gunawan, "Autonomous surface vessel for search and rescue operation," *Bulletin of Electrical Engineering and Informatics*, vol. 10, no. 3, pp. 1701–1708, 2021. [Online]. Available: https://www.beei.org/index.php/EEI/article/view/2599 (accessed May 17, 2024).

[10] H. Choi, C. Crump, C. Duriez, A. Elmquist, G. Hager, D. Han, F. Hearl, J. Hodgins, A. Jain, F. Leve, C. Li, F. Meier, D. Negrut, L. Righetti, A. Rodriguez, J. Tan, and J. Trinkle, "On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward," *Proceedings of the National Academy of Sciences*, vol. 118, no. 1, p. e1907856118, December 28th 2020. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1907856118 (accessed May 17, 2024).

[11] "Convention on the international regulations for preventing collisions at sea, 1972 (colregs)," International Maritime Organization. [Online]. Available: https://www.imo.org/en/About/Conventions/Pages/COLREG.aspx (accessed May 14, 2024).

[12] "Unity homepage," Unity Technologies, 2024. [Online]. Available: https://unity.com/ (accessed Mar. 1, 2024).

[13] *Manual: High Definition Render Pipeline overview*, Unity technologies, 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/index.html (accessed Feb. 28, 2024).

[14] A. de Tocqueville, R. Chapelain, V. Grimm, M. Muller, S. Lachambre, and A. Benyoub, "The new water system in unity 2022 lts and 2023.1," June 28th 2023. [Online]. Available: https://blog.unity.com/engine-platform/new-hdrp-water-system-in-2022-lts-and-2023-1 (accessed Jan. 1, 2024).

[15] "Ros-tcp-connector," Unity technologies. [Online]. Available: https://github.com/Unity-Technologies/ROS-TCP-Connector (accessed Dec. 1, 2023).

[16] "Ros homepage," Open Robotics, 2023. [Online]. Available: https://www.ros.org/ (accessed Mar. 1, 2024).

[17] E. Darles, B. Crespin, D. Ghazanfarpour, and J. Gonzato, "A survey of ocean simulation and rendering techniques in computer graphics," *Computer Graphics Forum*, vol. 30, no. 1, pp. 43–60, 2011. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00587242/file/survey.pdf (accessed Jan. 1, 2024).

[18] J. M. Bajo, G. Patow, and C. A. Delrieux, "Realistic Buoyancy Model for Real-Time Applications," *Computer Graphics Forum*, vol. 39, no. 6, pp. 217–231, 2020.

[19] M. Santos Pessoa de Melo, J. Gomes da Silva Neto, P. Jorge Lima da Silva, J. M. X. Natario Teixeira, and V. Teichrieb, "Analysis and comparison of robotics 3d simulators," in *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, 2019.

[20] E. Rohmer, S. P. N. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, November 2013, pp. 1321–1326. [Online]. Available: https://ieeexplore.ieee.org/document/6696520 (accessed May 17, 2024).

[21] "Gazebo homepage," Open Robotics, 2024. [Online]. Available: https://gazebosim.org/home (accessed Mar. 1, 2024).

[22] "Webots homepage," Cyberbotics, 2024. [Online]. Available: https://cyberbotics.com (accessed Mar. 1, 2024).

[23] "Argos homepage," 2024. [Online]. Available: https://www.argos-sim.info (accessed Mar. 1, 2024).

[24] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[25] "Advance your robot autonomy with ros 2 and unity," 2023. [Online]. Available: https://blog.unity.com/engine-platform/advance-your-robot-autonomy-with-ros-2-and-unity (accessed Jan. 25, 2024).

[26] "Autoferry gemini homepage," 2024. [Online]. Available: https://github.com/Gemini-team/Autoferry-Gemini (accessed Jan. 25, 2024).

[27] K. Vasstein, E. F. Brekke, R. Mester, and E. Eide, "Autoferry gemini: a real-time simulation platform for electromagnetic radiation sensors on autonomous ships," in *IOP Conference Series: Materials Science and Engineering*, vol. 929, no. 012032. IOP Publishing, 2020.

[28] *Manual: Unity Job System*, Unity Technologies, 2022. [Online]. Available: https://docs.unity3d.com/Manual/JobSystem.html (accessed Jan. 25, 2024).

[29] "Physx overview," NVIDIA. [Online]. Available: https://developer.nvidia.com/physx-sdk (accessed Feb. 28, 2024).

[30] *Manual: Built-in 3D Physics*, Unity Technologies, 2024. [Online]. Available: https://docs.unity3d.com/Manual/PhysicsOverview.html (accessed Feb. 28, 2024).

[31] *Scripting API: Rigidbody*, Unity Technologies, 2022. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rigidbody.html (accessed Feb. 28, 2024).

[32] "Shader," Khronos. [Online]. Available: https://www.khronos.org/opengl/wiki/Shader (accessed Apr. 11, 2024).

[33] *Manual: Modeling - Meshes - Primitives*, Blender Foundation. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html (accessed May 3, 2024).

[34] *Manual: Primitive and placeholder objects*, Unity Technologies. [Online]. Available: https://docs.unity3d.com/Manual/PrimitiveObjects.html (accessed Apr. 30, 2024).

[35] M. Omernick, *Creating the Art of the Game*. New Riders Pub, 2004.

[36] *Manual: Modeling - Meshes - Structure*, Blender Foundation. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/meshes/structure.html (accessed Apr. 30, 2024).

[37] *Maya Help: Modeling - Polygonal Modeling*, Autodesk Inc. [Online]. Available: https://help.autodesk.com/view/MAYAUL/2025/ENU/?guid=GUID-7941F97A-36E8-47FE-95D1-71412A3B3017 (accessed May 2, 2024).

[38] "Wavefront obj file format," Library of Congress. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml (accessed May 3, 2024).

[39] H. Gouraud, "Continuous shading of curved surfaces," *IEEE Transactions on Computers*, vol. C-20, no. 6, pp. 623–629, 1971.

[40] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, p. 192–198, jul 1977. [Online]. Available: https://doi.org/10.1145/965141.563893 (accessed May 17, 2024).

[41] H.-J. Ackermann and C. Hornung, "The Triangle Shading Engine," in *Eurographics Workshop on Graphics Hardware*, R. Grimsdale and A. Kaufman, Eds. The Eurographics Association, 1990.

[42] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, and M. Fernández, "Efficiently using connectivity information between triangles in a mesh for real-time rendering," *Future Generation Computer Systems*, vol. 20, no. 8, pp. 1263–1273, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X04000949 (accessed May 17, 2024).

[43] P. J. Besl, "Triangles as a primary representation," in *Object Representation in Computer Vision*, M. Hebert, J. Ponce, T. Boult, and A. Gross, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 191–206.

[44] M. Bern and D. Eppstein, "Mesh generation and optimal triangulation," in *Computing in Euclidean Geometry (Lecture Notes Computing)*, 2nd ed. World Scientific Pub Co Inc, 1992, pp. 23–90.

[45] P. Chou and T. Meng, "Efficient transmission of triangle meshes to graphics processors," in *2000 IEEE Workshop on SiGNAL PROCESSING SYSTEMS. SiPS 2000. Design and Implementation (Cat. No.00TH8528)*, 2000, pp. 275–284.

[46] "Blender - a 3d modelling and rendering package," Blender Foundation. [Online]. Available: http://www.blender.org (accessed Jan. 1, 2024).

[47] "3ds max," Autodesk Inc., 2024. [Online]. Available: https://www.autodesk.com/products/3ds-max/ (accessed Mar. 3, 2024).

[48] *Manual: About ProBuilder*, Unity Technologies. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.probuilder@6.0/manual/index.html (accessed May 2, 2024).

[49] "What's the difference between ray tracing and rasterization?" NVIDIA. [Online]. Available: https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/ (accessed May 6, 2024).

[50] J. Dischler and D. Ghazanfarpour, "A survey of 3d texturing," *Computers & Graphics*, vol. 25, no. 1, pp. 135–151, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849300001138 (accessed May 17, 2024).

[51] "What is uv mapping?" Pilgway. [Online]. Available: https://3dcoat.com/articles/article/what-is-uv-mapping/ (accessed May 14, 2024).

[52] "Texture coordinates (direct3d 9)," Microsoft Corporation. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d9/texture-coordinates (accessed May 14, 2024).

[53] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, 01 2009.

[54] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, May 2022. [Online]. Available: http://dx.doi.org/10.1126/scirobotics.abm6074 (accessed May 17, 2024).

[55] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974.

[56] W. Woodall, "Ros on dds." [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html (accessed Feb. 28, 2024).

[57] National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center, *Lidar 101: An Introduction to Lidar Technology, Data, and Applications*. Charleston, SC: NOAA Coastal Services Center, 2012, revised edition.

[58] D. L. Lu, "Orthographic projection of a registered point cloud," 2019. [Online]. Available: https://commons.wikimedia.org/wiki/File:Example.jpg (accessed Apr. 29, 2024).

[59] X. Yue, Y. Zhang, and M. He, "Lidar-based slam for robotic mapping: state of the art and new frontiers," 2023. [Online]. Available: https://arxiv.org/abs/2311.00276 (accessed May 14, 2024).

[60] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," 2017. [Online]. Available: https://arxiv.org/abs/1612.00593 (accessed May 14, 2024).

[61] Aivero, "Overview of depth cameras." [Online]. Available: https://aivero.com/overview-of-depth-cameras/ (accessed May 15, 2024).

[62] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004, ch. 9.1.

[63] A. N. (norro), "Epipolar geometry," 2007, cC BY-SA 4.0 license: https://creativecommons.org/licenses/by-sa/3.0/deed.en. Adaptations made: None. [Online]. Available: https://commons.wikimedia.org/wiki/File:Epipolar_geometry.svg (accessed May 14, 2024).

[64] S. Lee, "Depth camera image processing and applications," in *2012 19th IEEE International Conference on Image Processing*, 2012, pp. 545–548.

[65] RCraig09, "Time of flight," 2020, cC BY-SA 4.0 license: https://creativecommons.org/licenses/by-sa/4.0/. Adaptations made: None. [Online]. Available: https://commons.wikimedia.org/wiki/File:20200501_Time_of_flight.svg (accessed May 14, 2024).

[66] J. Geng, "Structured-light 3d surface imaging: a tutorial," *Advances in optics and photonics*, vol. 3, no. 2, pp. 128–160, 2011.

[67] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics & Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[68] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[69] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021. [Online]. Available: https://doi.org/10.21105/joss.02783 (accessed Mar. 7, 2024).

[70] P.-E. Sarlin, C. Cadena, R. Siegwart, and M. Dymczyk, "From coarse to fine: Robust hierarchical localization at large scale," in *CVPR*, 2019.

[71] P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperGlue: Learning feature matching with graph neural networks," in *CVPR*, 2020.

[72] K. A. Tsintotas, L. Bampis, and A. Gasteratos, "The revisiting problem in simultaneous localization and mapping: A survey on visual loop closure detection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 11, p. 19929–19953, Nov. 2022. [Online]. Available: http://dx.doi.org/10.1109/TITS.2022.3175656 (accessed May 17, 2024).

[73] O. Sørdalen, "Optimal thrust allocation for marine vessels," *Control Engineering Practice*, vol. 5, no. 9, pp. 1223–1231, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0967066197843614 (accessed May 17, 2024).

[74] A. Veksler, T. A. Johansen, F. Borrelli, and B. Realfsen, "Cartesian thrust allocation algorithm with variable direction thrusters, turn rate limits and singularity avoidance," in *2014 IEEE Conference on Control Applications (CCA)*, 2014, pp. 917–922.

[75] D. Halliday, R. Resnick, and J. Walker, *Fundamentals of physics: Extended*. John Wiley & Sons, 2005.

[76] T. C. Gillmer and B. Johnson, *Introduction to Naval Architecture*. Naval Institute Press, 1982.

[77] "Archimedes' principle," February 2024. [Online]. Available: https://www.britannica.com/science/Archimedes-principle (accessed Dec. 31, 2023).

[78] J. Kerner, "Water interaction model for boats in video games," February 27th 2015. [Online]. Available: https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games (accessed Jan. 1, 2024).

[79] D. Bernoulli, *Danielis Bernoulli ... Hydrodynamica, sive, De viribus et motibus fluidorum commentarii: opus academicum ab auctore, dum Petropoli ageret, congestum.* Sumptibus Johannis Reinholdi Dulseckeri, 1738. [Online]. Available: https://books.google.no/books?id=VP5xrx373N4C (accessed May 6, 2024).

[80] "Definition - hydrodynamics," Merriam-Webster. [Online]. Available: https://www.merriam-webster.com/dictionary/hydrodynamics (accessed Apr. 30, 2024).

[81] "Hydrodynamics," Encyclopedia Britannica. [Online]. Available: https://www.britannica.com/science/fluid-mechanics/Hydrodynamics (accessed Apr. 30, 2024).

[82] O. Darrigol, *Worlds of Flow: A History of Hydrodynamics from the Bernoullis to Prandtl.* Oxford University Press, 2005.

[83] "Conservation of energy," NASA. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/thermo1f.html (accessed May 6, 2024).

[84] "Bernoulli's equation," NASA. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/bern.html (accessed May 6, 2024).

[85] G. K. Batchelor, *An Introduction to Fluid Dynamics (Cambridge Mathematical Library).* Cambridge University Press, 2000.

[86] V. L. Streeter, *Handbook of Fluid Dynamics.* McGraw Hill Text, 1961.

[87] E. W. Weisstein, "Euler's equations of inviscid motion," Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/EulersEquationsofInviscidMotion.html (accessed May 8, 2024).

[88] "Viscosity," Encyclopedia Britannica. [Online]. Available: https://www.britannica.com/science/viscosity (accessed May 10, 2024).

[89] E. W. Weisstein, "Navier-stokes equations," Wolfram Research, Inc. [Online]. Available: https://mathworld.wolfram.com/Navier-StokesEquations.html (accessed May 9, 2024).

[90] O. Reynolds, "An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels," *Philosophical Transactions of the Royal Society*, vol. 174, p. 935–982, 01 1883. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rstl.1883.0029 (accessed May 17, 2024).

[91] "Reynold's number," NASA. [Online]. Available: https://www.grc.nasa.gov/www/k-12/airplane/reynolds.html (accessed May 10, 2024).

[92] "Ship hydrodynamics - design of the hull," Encyclopedia Britannica. [Online]. Available: https://www.britannica.com/technology/ship/Dynamic-stability#ref924191 (accessed Apr. 12, 2024).

[93] B. Baxter, *Baxter's Introduction to Naval Architecture*. Warsash Publishing, 1992.

[94] J. D. van Manen and P. van Oossanen, *Resistance*. Society of Naval Architects & Marine Engineers, 1988, ch. 5.

[95] H. Schlichting, *Boundary Layer Theory*, 6th ed. McGraw-Hill, 1968.

[96] L. Prandtl, "Motion of fluids with very little viscosity," in *Proceedings of the Third International Mathematics Congress*, Heidelberg, Germany, 1904.

[97] F. M. White, *Fluid mechanics*. McGraw-Hill, 1986.

[98] M. V. Dyke, *An Album of Fluid Motion*, 14th ed. Parabolic Press, Inc., 1982. [Online]. Available: https://courses.washington.edu/me431/handouts/Album-Fluid-Motion-Van-Dyke.pdf (accessed May 17, 2024).

[99] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[100] T. Fischer, W. Vollprecht, S. Traversaro, S. Yen, C. Herrero, and M. Milford, "A robostack tutorial: Using the robot operating system alongside the conda and jupyter data science ecosystems," *IEEE Robotics and Automation Magazine*, 2021.

[101] H. Kam, S.-H. Lee, T. Park, and C.-H. Kim, "Rviz: a toolkit for real domain data visualization," *Telecommunication Systems*, vol. 60, pp. 1–9, 10 2015.

[102] "Vcxsrv Windows X Server," Open Source Software, 2024. [Online]. Available: https://github.com/marchaesen/vcxsrv/ (accessed Jan. 15, 2024).

[103] "Conda: A package, dependency and environment management for any language," Anaconda, Inc., 2024, version 23.1.0. [Online]. Available: https://conda.io (accessed May 14, 2024).

[104] "Scripting in the Water System," Unity Technologies, 2023. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/WaterSystem-scripting.html (accessed Sept. 21, 2023).

[105] S. W. Zucker and R. A. Hummel, "A three-dimensional edge operator," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-3, no. 3, pp. 324–331, 1981.

[106] O. Kaleniuk, *Geometry for programmers*. Manning Publications, 2023.

[107] R. F. Graf, *Modern Dictionary of Electronics, 7th Edition*. Newnes, 2023.

[108] W. M. Kelley, *Calculus I (Idiot's Guides)*. Alpha, 2016.

[109] J. F. Hughes, A. van Dam, M. McGuire, D. Sklar, J. Foley, S. Feiner, and K. Akeley, *Computer graphics: principles and practice*, 3rd ed. Upper Saddle River, New Jersey: Addison-Wesley, 2014.

[110] *Scripting API: Transform.TransformPoint*, Unity technologies. [Online]. Available: https://docs.unity3d.com/ScriptReference/Transform.TransformPoint.html (accessed Feb. 28, 2024).

[111] S. of Naval Architects, M. E. U. Technical, and R. C. H. Subcommittee, *Nomenclature for Treating the Motion of a Submerged Body Through a Fluid: Report of the American Towing Tank Conference*, ser. Technical and research bulletin. Society of Naval Architects and Marine Engineers, 1950. [Online]. Available: https://books.google.no/books?id=VqNFGwAACAAJ (accessed May 17, 2024).

[112] A. Weber, B. Jenny, M. Wanner, J. Cron, P. Marty, and L. Hurni, "Cartography meets gaming: Navigating globes, block diagrams and 2d maps with gamepads and joysticks," *The Cartographic Journal*, vol. 47, no. 1, pp. 92–100, 2010. [Online]. Available: https://doi.org/10.1179/000870409X12472347560588 (accessed Jan. 5, 2024).

[113] A. H. Cummings, "The evolution of game controllers and control schemes and their effect on their games," in *The 17th Annual University of Southampton Conference*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:14629263 (accessed Jan. 1, 2024).

[114] *Manual: Input System*, Unity Technologies, 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Actions.html (accessed Mar. 4, 2024).

[115] *Scripting API: Raycast*, Unity Technologies. [Online]. Available: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (accessed May 14, 2024).

[116] *Manual: HDRP Camera*, Unity Technologies, 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/HDRP-Camera.html (accessed Feb. 28, 2024).

[117] "Depth test," Khronos Group. [Online]. Available: https://www.khronos.org/opengl/wiki/Depth_Test (accessed May 13, 2024).

[118] *Manual: Custom Pass*, Unity Technologies. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/Custom-Pass.html (accessed Feb. 27, 2024).

[119] *Scripting API: RenderDepthFromCamera*, Unity Technologies. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@15.0/api/UnityEngine.Rendering.HighDefinition.CustomPassUtils.RenderDepthFromCamera.html (accessed Feb. 27, 2024).

[120] K. Conley, "Ros enhancement proposal." [Online]. Available: https://ros.org/reps/rep-0001.html (accessed Dec. 1, 2023).

[121] A. E. Conrady, "Decentred lens-systems," *Monthly Notices of the Royal Astronomical Society*, vol. 79, no. 5, pp. 384–390, March 1919. [Online]. Available: https://doi.org/10.1093/mnras/79.5.384 (accessed Mar. 3, 2024).

[122] P. G. Vivo and J. Lowe, "Random - the book of shaders," 2015. [Online]. Available: https://thebookofshaders.com/10/ (accessed Mar. 4, 2024).

[123] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[124] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1330–1334, December 2000, mSR-TR-98-71, Updated March 25, 1999. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-flexible-new-technique-for-camera-calibration/ (accessed May 17, 2024).

[125] Q. Kong, T. Siauw, and A. Bayen, *Python Programming and Numerical Methods: A Guide for Engineers and Scientists*. Elsevier, 2020, chapter 20.1. [Online]. Available: https://shop.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9 (accessed May 17, 2024).

[126] R. Borase, D. Maghade, S. Sondkar *et al.*, "A review of pid control, tuning methods and applications," *International Journal of Dynamics and Control*, vol. 9, pp. 818–827, 2021. [Online]. Available: https://doi.org/10.1007/s40435-020-00665-4 (accessed May 14, 2024).

[127] T. A. Johansen and T. I. Fossen, "Control allocation—a survey," *Automatica*, vol. 49, no. 5, pp. 1087–1103, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0005109813000368 (accessed May 17, 2024).

[128] R. Penrose, "A generalized inverse for matrices," *Proceedings of the Cambridge Philosophical Society*, vol. 51, no. 3, pp. 406–413, 1955.

[129] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics YOLO Version 8.0.0," Ultralytics, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics (accessed Apr. 10, 2024).

[130] Unity Technologies, "Unity Perception package," 2020. [Online]. Available: https://github.com/Unity-Technologies/com.unity.perception (accessed May 8, 2024).

[131] "Navier usn homepage," 2024. [Online]. Available: https://navierusn.com/ (accessed May 8, 2024).

[132] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The marathon 2: A navigation system," *CoRR*, vol. abs/2003.00368, 2020. [Online]. Available: https://arxiv.org/abs/2003.00368 (accessed May 17, 2024).

[133] R. Agrawal, "Teleop twist keyboard for ros2," 2024. [Online]. Available: https://github.com/rohbotics/ros2_teleop_keyboard (accessed Mar. 26, 2024).

[134] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[135] "Cesium homepage," 2024. [Online]. Available: https://cesium.com/ (accessed Apr. 12, 2024).

[136] "Photorealistic 3d tiles," Google. [Online]. Available: https://developers.google.com/maps/documentation/tile/3d-tiles (accessed Apr. 12, 2024).

[137] "Iso 10303-21:2016 - industrial automation systems and integration," International Organization for Standardization. [Online]. Available: https://www.iso.org/standard/63141.html (accessed May 6, 2024).

[138] C.-W. Liu, H. Wang, S. Guo, M. J. Bocus, Q. Chen, and R. Fan, *Stereo Matching: Fundamentals, State-of-the-Art, and Existing Challenges*. Singapore: Springer Nature Singapore, 2023, pp. 63–100. [Online]. Available: https://doi.org/10.1007/978-981-99-4287-9_3 (accessed May 17, 2024).

[139] "Wam-v github page," 2023. [Online]. Available: https://github.com/osrf/vrx/tree/main/vrx_urdf/wamv_description (accessed Dec. 31, 2023).

[140] "Prototype textures," February 2024. [Online]. Available: https://kenney.nl/assets/prototype-textures (accessed Dec. 31, 2023).