CataBotSim: A Realistic Aquatic Simulator for Autonomous Surface Vehicle Testing

Dinithi Silva-Sassaman
Department of Computer Science
Dartmouth College
Hanover, USA
0009-0000-7431-4799

Mingi Jeong
Department of Computer Science
Dartmouth College
Hanover, USA
0000-0002-9097-6343

Paul Sassaman
Independent
Hanover, USA
0009-0008-1000-6568

Zitong Wu
University of North Dakota
Grand Forks, USA
0009-0006-4281-9411

Alberto Quattrini Li
Department of Computer Science
Dartmouth College
Hanover, USA
0000-0002-4094-9793

***Abstract*—While there are many self-driving car simulators, there is a general lack of high-fidelity aquatic simulators that allow testing perception and navigation pipelines for autonomous surface vehicles (ASVs). We present an open source, user friendly, and realistic Unity3D-based aquatic simulator for ASVs, called *CataBotSim*, which includes support for multiple ASV models, on-board sensors, such as LiDAR, Marine Radar, IMU, GPS, and cameras, as well as a number of different objects that can act as static or dynamic obstacles. To achieve high-fidelity simulations, we incorporated environmental effects like rain, currents, and wind, which impact sensor measurements and/or ASV motions as observed in real-world environments. We included interfacing with the Robot Operating System (ROS). We also provided tools to generate real-world scenes, create datasets, record navigation metrics, and provide ground truth perception data to ease robotic simulation experiments. The project code for the CataBotSim can be found at https://github.com/dartmouthrobotics/catabot_unity/. A demonstration video of CataBotSim is available at https://youtu.be/Rgfkm352ho4.**

I. Introduction

This paper presents a realistic simulation platform and tools that can support the development and testing of Autonomous Surface Vehicle (ASV) perception and navigation pipelines, under a number of different external disturbances, sensor noises, and scanned environments from Earth currently not present in other available simulators (Fig. 1).

Our simulator includes water dynamics, wind physics, and rain. We added a twin-hull ASV model controllable with two motors with several sensors typically on-board ASV platforms – Inertial Measurement Unit (IMU), camera, marine radar, GPS, and LiDAR – with noise. We also added tools for perception segmentation, terrain generation, and automatic scenario generation with up to 100 additional ASVs.

These features augment what is currently present in the literature and will contribute in advancing ASV's autonomous navigation, which is key to subsequent operations with high societal impact, such as environmental monitoring and autonomous transportation [1].

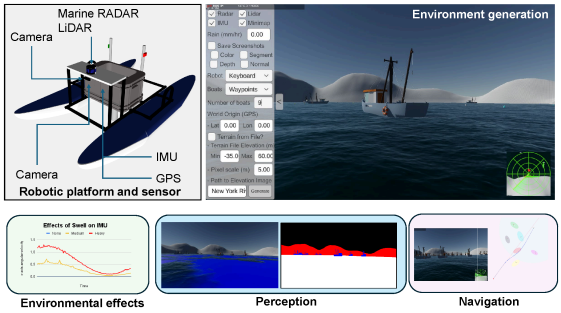

Fig. 1: CataBot Aquatic Simulator Environment developed in Unity3D.

Realistic simulators are necessary for the development of algorithms, especially in areas where conducting experiments is expensive due to high logistics overhead, high-cost equipment, and difficulty in testing in many different scenarios, as is common for ASVs. Moreover, reliable simulators play a key role for validating the safety of navigation and collecting fit-for-purpose data, which is critical for developing and testing autonomy. While there are a number of simulators for ground robots [2], self-driving cars [3], [4], and aerial drones [5], the aquatic surface simulators are comparatively more limited to the simulators mentioned in their ability to replicate realistic environmental and sensor conditions [6].

This paper contributes to enhancing the realism of ASV simulators, extending its applicability across a broad spectrum of real-world conditions. The key contributions in this paper can be categorized as follows:

- *High-fidelity and robust simulator implementation*: Utilizing Unity3D [11] for simulator development to ensure the flexibility in creating diverse realistic environments with

This work is supported in part by NSF CNS-1919647, 2144624, OIA1923004 and NH Sea Grant.

TABLE I: Comparison of simulators including surface and other domains. Our simulator, compared to existing ASV simulators, includes marine radar, weather effects affecting the sensor, and the ability to run a high number of robots.

	environment	**groundtruth data capability**	**sensors**[1]	**physics (hydrostatic)**	**weather (rain)**	**tested total number of agents**[2]	**base platform**
Gazebo [7]	General	N	L, C, I, G	Y	N	C	-
VRX [8]	Surface	N	L, C, I, G	Y	N	C	Gazebo
AirSim [5]	General without marine	Y	L, C, I, G	N/A	Y	B	Unreal Engine
CARLA [3]	Urban	Y	L, C, I, G, R	N/A	Y	B	Unreal Engine
MARUS [9]	Marine[3]	Y	L, C, I, G	Y	N	Unknown	Unity
USV Sim [6]	Surface	N	I, G	Y	N	C	Gazebo
AMSVS [10]	Surface	N	L, C, I, G	Y	N	Unknown	Gazebo
CataBotSim	**Surface**	**Y**	L, C, I, G, **MR**	**Y**	**Y**	**A**	Unity

[1]: supported sensor for detection and ego motion. L: LiDAR, C: camera, I: IMU, G: GPS, R: Radar, MR: Marine Radar.
[2]: the maximum number of of deployable agents that can supports other tasks, e.g., perception. A: > 20, B: $10 - 20$, C: < 10.
[3]: including both underwater and surface.

high fidelity in 3D geometry and physics simulations;

- *Realistic sensor simulation*: Integrating a variety of sensors for ASV navigation, including realistic noise effects from environmental conditions (such as rain in LiDAR sensors);
- *Applicability for marine autonomy*: Showcasing the simulator's capabilities through various scenarios, highlighting its utility for training perception pipelines and validating navigation planning as a core part of ASV autonomy; and
- *Integration with middleware and open-sourcing*: Seamless integration with the Robot Operating System (ROS) [12], thereby facilitating the incorporation of existing algorithms into the simulation environment. Additionally, we make the simulator open-source for the community.

While additional realistic effects (e.g., bias) will be added in the future, this first enhancement of ASV simulators will contribute towards a faster ASV technology progress, necessary for a large variety of important operations, such as marine transportation and algal bloom monitoring.

II. Related Work

Simulations have been widely used in robotics. An in-depth survey of simulators by Collins et al. [13] explores different physics based simulations for different areas in robotics such as manipulators, medical robots, and soft robotics, as well as navigation on land, aerial, and marine environments. The authors further divide the marine robotics simulators into two categories: simulators for surface vehicles and simulators for underwater robots. Here we specifically discuss the simulators in terms of capabilities provided for ASV simulation – see Table I, showing the main characteristics being compared in a number of popular simulators and our proposed simulator.

There are general robotics simulators, appeared in the early 2000s, first focusing on ground robots and then expanding to other platforms. Notably, Gazebo [7] is one of the most popular 3D simulation platforms used in robotics given its integration with ROS. However, in its standard form, the visual realism is limited, especially for water and grass. Fluid dynamics simulation was also not present until Gazebo was extended to support ASVs with the Virtual RobotX (VRX) [8]. The supported sensors include the ones already provided for ground robots, i.e., camera, LiDAR, IMU, and GPS. There is a limit in number of robots that can be supported in real time given the physics engine used in Gazebo. In CataBotSim we provide a water simulator that has physics, buoyancy, and configurable wind physics to make the testing environment more realistic. Moreover, CataBotSim is integrated with simulated marine radar and heavy maritime traffic.

More recently, game engines have become the base to develop robotic simulators due to the visual realism they can provide. An early robot simulator was USARSim [14], which is open source and built with Unreal Engine [15] for both research and educational purposes. USARSim is designed to be extended by users to create custom virtual robots. While the simulator has an extensive set of sensors and actuators integrated into the system, it lacks realistic hydro-dynamics since it was greatly simplified for simulation purposes. Additionally, USARSim is not designed to simulate ASVs. In 2017, Shah et al. introduced AirSim [5], an open source physics based flight simulator made with Unreal Engine, which mimics a quadcopter in different environments. AirSim uses computer vision techniques to annotate the virtual environments in real time with color coded object segmentation. Similar to AirSim, CataBotSim also provides the user with the option to annotate the images through the Unity Perception Package [16]. Even though AirSim is a promising start for a flight simulator, it has limitations such as an absence of object detection, raycasting, and a lack of realistic noise models. This could potentially makes it difficult to gather realistic data. To eliminate these setbacks, CataBotSim has collision based physics and an efficient raycasting used in the simulated camera for LiDAR. Furthermore, we added Gaussian noise to its IMU, marine radar, and LiDAR sensor models to give more realistic data. Another difference worth mentioning between the two systems is that AirSim only provides a single fictional location by default while CataBotSim provides a way to recreate physical environments from elevation maps. This increases the number of available environments to include any location on Earth.

From the advances in self-driving cars, CARLA [3] is an open source driving simulator built with Unreal Engine 4.0 to create content for testing and validating autonomous driving approaches. CARLA provides different weather conditions such as a sunny day, daytime rain, daytime shortly after rain, and a clear sunset, similar to how our CataBotSim provides adjustable weather conditions such as wind, rain, and sunlight. Both systems also provide ground truth semantic segmentation and depth segmentation, allowing users to represent various

datasets in various weather conditions and different environments so they can train and validate machine learning models. Another driving simulator worth mentioning is LGSVL [4] which is an open source application implemented using the Unity3D game engine. Very similar to CataBotSim, LGSVL has five physical sensors (LiDAR, Radar, IMU, GPS, and Camera) and four virtual sensors (2D Ground Truth, 3D Ground Truth, Depth camera, and Segmentation camera). LGSVL is also integrated with autonomous driving systems such as Autoware and Apollo. Additionally, LGSVL gives the user the freedom to customize their modules as well as customize sensors, which is also an option in our system CataBotSim. These simulators for self-driving car scenarios inspired us to use Unity for ASVs.

There have been a few underwater simulators based on Unreal or Unity, such as MarineSIM [17] and HoloOcean [18]. While their realism is enhanced thanks to the use of the game engines, they cannot be directly used for water surface simulation given the different various unique physics forces interacting on the surface robots, such as wind and water waves and currents.

As for realistic sensor simulations, Unity3D does offer a sensor SDK [19]. However, it is currently only available through a costly paid licence (approximately $450/mo). Additionally, even though there are several LiDAR models available under this SDK, it lacks LiDAR interference with rain. CataBotSim has accurate LiDAR interaction with different rain conditions available for free.

In summary, there are a limited number of simulators focusing on Autonomous Surface Vehicles (ASVs) ([6], [8]–[10]) compared to the plethora available in other domains (e.g., [3], [5], [7], [20]). Gazebo is predominantly used in the marine domain ([6], [8], [10]), with the exception of [9]. While integration with ROS at the development stage has expanded their application, these simulators exhibit limitations in some aspects of realism: (1) *less photo-realistic visualization*: Gazebo's visual effects fall short of Unity-based simulators, posing challenges in bridging the simulation-to-reality gap [21]; (2) *constrained by real-time factor*: particularly when deploying multiple agents in a simulated environment, the simulators struggle to operate in real-time. This computational burden complicates the execution of crucial tasks such as object detection and planning, necessary for real robots; (3) *lack of groundtruth generation*: although they allow for testing realistic motion of in-water objects including ASVs, the inability to support dataset generation with ground truth restricts their utility in validating perception tasks. Consequently, this limitation hinders a comprehensive evaluation of the overall autonomy pipeline. We aim at solving these limitations with our simulator.

III. CataBotSim: A modular multi-ASV simulator

We propose a multi-ASV simulator built with Unity3D, which can accurately simulate various surface terrain types at different elevations; various aquatic surface, wind, and maritime traffic conditions; marine radar, LiDAR, and IMU sensors; and manual or autonomous control via ROS.

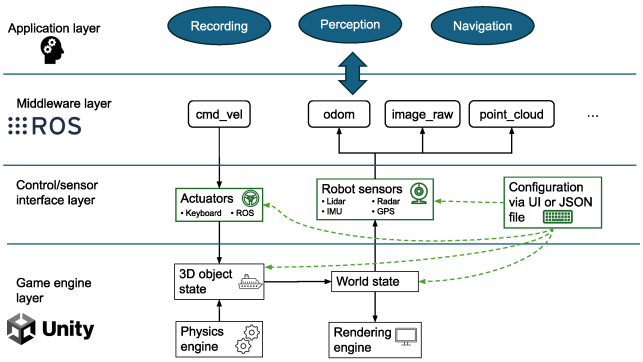

Fig. 2: CataBotSim's architecture.

The advantage to using Unity3D is in its realism for rendering 3D scenes and performing accurate physics by default. Unity also provides a ROS interface layer to easily connect the simulator's C# scripts with external software.

Each aquatic vehicle in the simulator, whether it's the primary robot or simulated traffic, can be controlled directly by the simulator or by an external ROS application. Each vehicle also continuously broadcasts its state (position, sensor values, etc) over ROS, allowing external software to track what is happening in the simulator. This results in an interface that provides seamless integration with other robotics use cases.

The simulator also contains an extensive settings menu, allowing the user to granularly adjust features like active sensors, what data are being recorded, how aquatic vehicles should be controlled, and what the terrain will look like. This settings menu can be adjusted at runtime, but it can also be configured with a JSON file beforehand, allowing automated simulation runs.

The overview of the architecture can be found in Fig. 2. Our simulation framework is composed of four main components described in the following: 1) Physics simulation and scene rendering of aquatic surface environments, 2) Robot model and sensors (LiDAR, Marine Radar, IMU) with optional transmission to ROS, 3) Simulation of maritime traffic, and 4) tools that can help users in easily setting up multiple simulations and record session data.

A. Physics simulation and rendering

1) 3D Environments: Unity allows users to import 3D models of scanned physical environments pulled from external sources like ArcGIS [22], and they can be inserted into the simulation with minimal effort. Unity also has robust tools for creating custom terrain from scratch, allowing users to simulate conditions that may not be easily accessible from physical scans.

To simplify the creation of 3D environments by the user, we have created a tool that combines these two elements. The user is able to import a grayscale elevation image from external sources, configure elements like lowest and highest

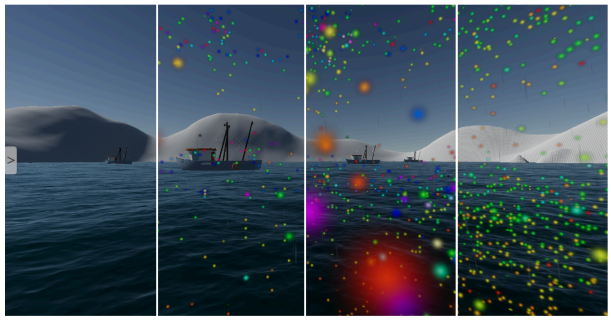

Fig. 3: The user can change the rain intensity via the program UI. Different rain intensities and their effects on the LiDAR sensor are shown from left to right - 0, 10, 25, and 50 mm/hr.

Fig. 4: The radar data are visualized through a minimap graphic in the bottom right of the screen. The green beam indicates the radar's current position, and the green dots are radar returns.

elevation and GPS coordinates, and then generate a 3D terrain object. This option is more accessible to the general public as many mapping services do not provide the ability to download 3D models for offline use without specific software or subscriptions.

2) Water surface simulation: We added a robust and highly configurable water surface simulation [23], which is available as an official extension to the Unity3D engine. This includes physical elements like swell and current, as well as visual elements like smoothness, foam, and refraction. Unity3D also has a robust physics engine. This allows us to simulate buoyancy for floating objects, making them bob and tilt with the movement of the water's surface. Users can also create custom current maps which affect water visuals and floating object physics using this extension. Note that objects, like boats, on the water surface could hit the bottom if the area is too shallow. Different bottom materials with different collision behaviors will be included in the future.

3) Wind: We also added wind physics, which accurately simulate how various intensities and directions of wind affects the robot, boats, and other floating bodies. Wind can be set as a global force, but it is also possible to set multiple arbitrarily shaped zones where the wind blows in a different direction or at a different strength. Currently, wind is not automatically altered by the presence of physical objects like terrain or buildings. Users must manually set the wind patterns, similar to how water current maps work.

4) Rain: We have incorporated and modified a configurable rain effect using [24], a free rain module from the Unity Asset Store. This includes: 1) a visual effect that impacts camera detection, and 2) a reflective effect that influences the intensity of LiDAR measurements (Section III-B). Examples of various rain intensities are presented in Fig. 3.

B. Robot model and sensors

Robots can be configured according to different motion models and equipped with various sensors.

1) Robot Model: A configuration file defines the robot 3D model and the included thrusters. Currently, a monohull and a catamaran style robots are provided together with our simulator.

We provide a controller that allows the user to move the robot, either directly from the simulator interface or via ROS velocity messages.

2) Simulation of sensors: There are a number of sensors we implemented that can be onboard the simulated robot, including LiDAR, Marine Radar, and IMU. These sensors are visually represented in the 3D environment, allowing users to see how the sensors are interacting with nearby items in the environment. Additionally, the sensor data are represented in a minimap (a small top down view of the world around the robot, as shown in Fig. 4) allowing for a broader visual understanding of the sensor results. The sensor data can also be optionally transmitted as raw data over ROS topics.

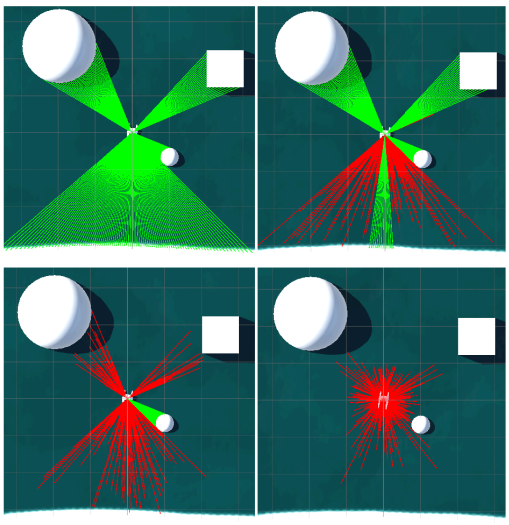

Fig. 5: LiDAR is affected by rain intensity, specifically with higher intensities resulting in lower visibility on average. Red lines indicate LiDAR beams that have been "intercepted" by a raindrop before they reach an object, while green lines successfully reached an object.

a) LiDAR: The LiDAR sensor simultaneously captures measurements within at a set angular resolution (45 degrees and 32 readings vertical, 360 degrees and 512 readings horizontal) and maximum range (100 meters), at around 5 Hz, e.g., matching the capabilities of the OS1 [25] LiDAR sensor. We accomplish this efficiently by using the Pixel Position

camera from Unity's Perception Package. One camera pixel's information is the equivalent of one raycast, allowing us to capture hundreds of readings per frame. While this allows the simulator to capture completely accurate ground truth data, the Unity engine does not provide the ability to modify the depth capture with elements like rainfall by default. With this in mind, we have added some additional equations based on the work of [26], [27], and [28] to simulate attenuation due to collisions with raindrops, as shown in Fig. 5. Since performing these calculations on the CPU would reduce the framerate of the simulation to approximately 2 frames per second, we instead run them through a compute shader on the GPU, thereby keeping the framerate above 30fps.

The steps are as follows.

Step one: Calculate the number of drops, n, that would probabilistically be in a LiDAR beam of the given length and diameter. We can model the average number of drops in a cubic meter with

$$N(D) = 8000 \cdot \exp(-4.1 \cdot R^{-0.21} \cdot D) \tag{1}$$

where R is the rate of rainfall in mm/hr and D is average drop diameter (about 3.25 mm) [26], [28].

Next, we need to determine the average number of raindrops in the entire beam. Since we have the distance Z to the object from the raycast and the radius of the beam r, this is:

$$\mu = \pi \cdot r^2 \cdot Z \cdot N(D) \tag{2}$$

To make sure this is accurate, we use a Poisson distribution to finalize the number of drops:

$$p(n) = \exp(-\mu) \cdot (\mu^n / n!) \tag{3}$$

where, as stated earlier, n is the number of drops being predicted to be in the beam [26].

Step two: Calculate the Returned Intensity of the Object (RIO) and the Returned Intensity of the Rain Drop (RIRD). The relative power equation (a modified version of [26](2) as discussed in [26]'s section III.A) that will be used for both values is

$$P_n(z) = \frac{\rho}{Z^2} \cdot \exp(-2 \cdot \alpha \cdot Z) \tag{4}$$

where ρ is the backscatter coefficient and α is the scattering coefficient [26].

Let's start with RIO. For a diffusely reflecting surface, the backscattering coefficient ρ is

$$\rho = \frac{s}{\pi} \tag{5}$$

where s is the amount of reflection (e.g., a 90% diffusely reflecting surface should have a value of $\rho = \frac{0.9}{\pi}$) [27].

The scattering coefficient α is:

$$\alpha = \frac{n \cdot Q_{ext}(D, \lambda) \cdot (\frac{\pi \cdot D^2}{4})}{\pi \cdot r^2 \cdot Z} \tag{6}$$

where $Q_{ext}(D, \lambda)$ is the extinction efficiency for the raindrops for a given wavelength (λ) and drop diameter [28]. [28] also shows that for $\lambda = 905 \cdot 10^{-9} m$ (the most common wavelength for surface LiDARs) and our average drop diameter 3.25 mm, $Q_{ext}(D, \lambda)$ is approximately 2.

RIRD returns the intensity of the first raindrop hit by the beam. Considering this, α is 0 since there cannot be additional raindrops before the first raindrop. Additionally, the backscatter equation for the raindrop is similar to the extinction coefficient for the raindrops in RIO:

$$\alpha = \frac{n \cdot Q_{back}(D, \lambda) \cdot (\frac{\pi \cdot D^2}{4})}{\pi \cdot r^2 \cdot Z} \tag{7}$$

where $Q_{back}(D, \lambda)$ is the backscatter coefficient for a given wavelength and drop diameter, and Z is the distance to the first raindrop [28]. [28] once again shows that $Q_{back}(D, \lambda)$ is approximately 2 for the previously given parameters. Z is calculated by taking n random distances within the range of the beam, and keeping the smallest distance.

Step three: Calculate the returned distance. We use [26]'s logic flow to determine the final returned distance. If the raycast hit something and there were no raindrops in the beam, return the distance to the object. If there was at least one raindrop in the beam, compare RIO to RIRD. If RIO is the larger value and is bigger than a detection threshold ([26] sets this as $0.5 \cdot 10^{-5}$), return the distance to the object. If RIO was less than the threshold, return no collision. If RIRD was larger than RIO or the raycast didn't hit anything, we compare RIRD to the threshold instead. If RIRD was bigger than the threshold, return the distance to the raindrop. Return no collision otherwise.

b) Marine Radar: Differently from the current simulators, we implemented a key exteroceptive sensor in the marine domain, i.e., Marine Radar. Marine Radar is similar to LiDAR in that the distance is calculated with a raycast. In accordance with real-world sensor characteristics, we modeled the Marine Radar's scanning frequency to be slower (0.5 Hz) than that of the LiDAR (2 Hz to 4 Hz) for a complete 360-degree sensor measurement. We also apply Gaussian noise to this distance reading instead of calculating rain interference, as radar is generally not affected by rain as the LiDAR.

c) Inertial Measurement Unit: Our IMU sensor takes the ground truth 6DOF from Unity, but we also added Gaussian noise to the measurements to make the sensor values more realistic. We will add bias in the future. Example measurements in varying amounts of swell can be seen in Fig. 6.

d) GPS: For more precision, instead of using a projection system (e.g., UTM), GPS latitude and longitude (*lat*, *lon*) in meters (lat_m, lon_m) is calculated with the help of lookup tables from [29] containing the size of each degree of *lat* and *lon* as a length in meters (lat_{height_m}, lon_{width_m}). The biggest difficulty comes from how the distance between the degrees changes with each increase in latitude, but this is solved by preemptively totaling up the distance to the start of each degree of latitude and storing it in a third lookup table (lat_{start_m}).

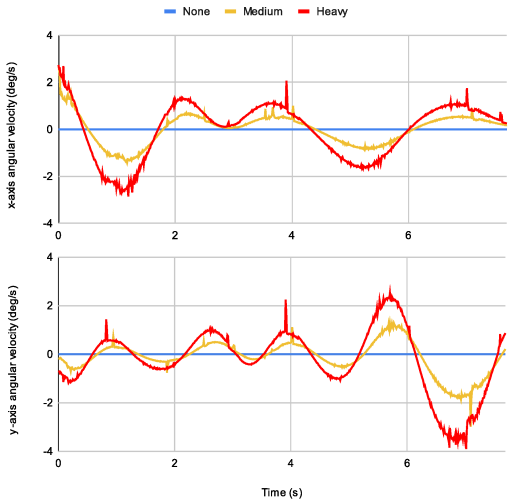

Fig. 6: Angular velocity over time with varying amounts of swell (amplitude – Medium: 0.62 m, Heavy: 1.25 m).

$$lat_m = lat_{start_m}[\text{floor}(lat)] + lat_{height_m}[\text{floor}(lat)] \cdot (lat\%1) \quad (8)$$

$$lon_m = \text{lerp}(lon_{width_m}[\text{floor}(lat)], lon_{width_m}[\text{ceil}(lat)], lat\%1) \cdot lon \quad (9)$$

where lerp is linear interpolation.

C. Simulation of maritime traffic

We have added the ability to simulate various types of maritime traffic with varying shapes and sizes of boats (with the default being a single type of boat). These boats can be controlled by the simulation directly or they can be controlled externally through standard ROS messages.

We added two main primitive behaviors to control each boat as part of the maritime traffic. The first option is to move a boat in a circle with a set radius. The alternative is to have the boat follow a path made of an arbitrary set of points (visually represented in the simulation by floating buoys). These points can be manually set or automatically generated, producing consistent or random obstacles as needed.

If an external package needs to control the position of the boats, there are two options available here as well – a low-level control via velocity messages or a high-level control via waypoints. We also provide the option to broadcast each boat's state in the simulator through both the Odometry and transformations TF topics to any external application that needs the data.

D. Useful Tools

1) Data recording: In addition to transmitting sensor data via ROS topics, the simulator can record everything to a csv file in a time synchronous manner, which allows for review and logging at a later time. The simulator can also be configured to record a set of screenshots (color, depth, normals, segmentation, or any combination of the above as seen in Fig. 7) multiple times per second, helping learning-based pipelines in creating datasets. It is also possible to record

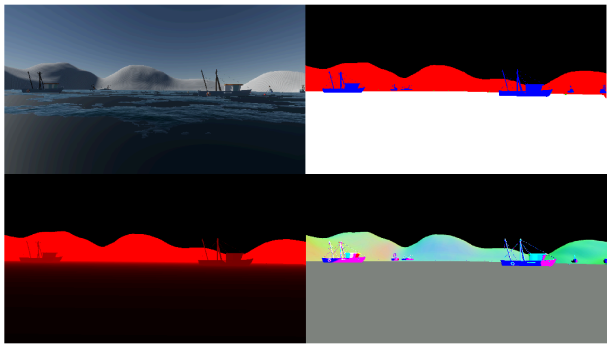

Fig. 7: Unity's perception package allows for the recording of ground truth data from the camera. This includes variations like color (top left), segmentation (top right), depth (bottom left), normals (bottom right), bounding box, and more.

a single color screenshot of the simulator's screen at any time by hitting the Space key. We have also added the option to stop perception at the water's surface (as seen in Fig. 7) or to allow perception to reach the bottom of the body of water.

Note that because the simulator is integrated with ROS, 'rosbag' can also be used for recording the data.

IV. Setting up and Running the Simulation

Creating and running a simulation is simplified thanks to the tools we provide. Here we describe the workflow.

A. Creation of an environment

Creating a terrain object from an elevation image is accomplished in three steps: a) set the path to a grayscale elevation image in the settings UI, b) adjust a few positioning settings, and c) click the Generate button. The results will be similar to Fig. 8. Elevation images can be found in a wide variety of locations, but we recommend using Esri's official "TopoBathy" layer in ArcGIS Online [22] as it provides a dynamic range adjustment option that automatically scales the data between the local minima and maxima within the current viewport.

There are a few assumptions to keep in mind when setting up a 3D environment from an elevation image.

The first is that the water will by default always be represented as an infinite "plane". At runtime, the water will distort vertically to represent swell, but it is effectively a plane. We

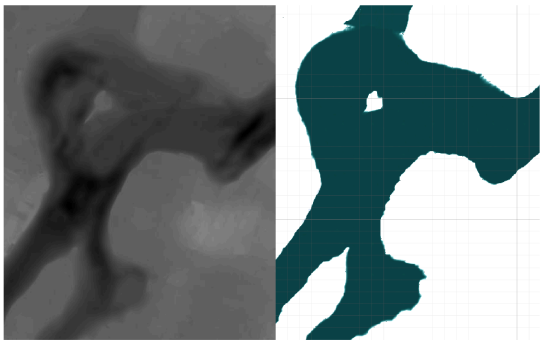

Fig. 8: An elevation image (top left) can be converted into a Unity Terrain (top right).

have also set the world height of the water to 0 for simplicity and to reduce the likelihood of floating point accuracy loss. This means that if a water source is not at sea level (e.g., most inland water), elements like terrain elevation min and max will have to be adjusted accordingly. Min will always represent how many meters the lowest point in the elevation image is below the water's surface level, and Max represents how many meters the highest point is above the water.

The second assumption is in how world position is handled. The terrain will be generated so that the bottom left corner of the elevation image is aligned with the origin of the virtual world. This means that all locations visible within the image will be located at positive coordinates in the virtual world. If the GPS coordinates of the bottom left corner are known, this can also be set within the settings UI to provide accurate GPS positioning of the robot within the virtual world.

B. Configuring the robot

Once the environment configuration is finalized, the user can launch the simulator by setting the initial position of the ASV and pressing the start button, which enables manual control of the ASV. Any additional configuration (e.g., enabling sensors, setting the rainfall amount, spawning random boats) can be accomplished through the settings menu. The default settings can also be changed by altering the file SettingsConfig.json allowing the simulation to start in an automated way.

V. Demonstrations for Perception and Navigation

We demonstrate how the simulator can aid ASV perception and navigation pipelines. All testing was performed on an Intel i9 desktop with an NVIDIA RTX 4070 graphics card and 64 GB of RAM. Most testing was performed in Windows 11 with the exception of the Navigation tests which were done in Ubuntu 18.04 to test with ROS.

A. Simulator Performance

Our proposed simulator can run in real-time, and the real-time factor remains stable even when a large number of boats are deployed simultaneously, as demonstrated in Table II. We measured frame rates at different combinations of the number of spawned boats in the simulator (0-100) and the active perception package elements (Segmentation, Depth, Normals, and unaltered Color screenshots).

TABLE II: Comparison of Frames per Second as per use of perception packages in our simulator. LiDAR (which also uses perception) is always active.

Number of Boats	Frame Rates per Second		
	With no perception package active	With Segmentation recording enabled	Four perception packages active
0	55	33	15
5	54	32	14.5
10	52	30	14
20	51	29	13.5
50	36	28	13
100	32	24	12

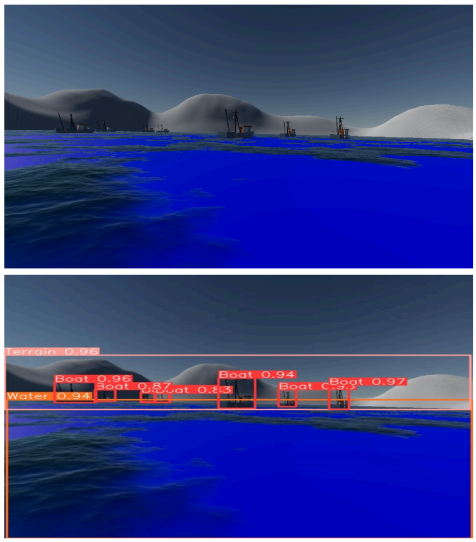

Fig. 9: CataBotSim's capability of the perception task application. (top) raw image generated in our CataBotSim environment including boats, terrain, and water. (bottom) inference result based on the trained state-of-the-art model.

B. Perception

Users can utilize images and ground truth labels generated by CataBotSim for the training and testing of perception tasks, such as object detection. We employed the state-of-the-art YOLOv8 model [30] because of its rapid and precise inference capabilities, which are highly beneficial for object detection in the maritime domain. We formatted the ground truth to meet the training and validation methods' requirements. To enhance the diversity of data derived from CataBotSim-generated images, we implemented augmentation strategies including flipping, cropping, and adding noise.

As shown in Fig. 9, the state-of-the-art model effectively detects objects, such as boats, from an egocentric view (inference time approximately 20 ms). CataBotSim's perception capabilities are crucial for testing object tracking and the integration of heterogeneous sensors, accommodating sensing models and incorporating realistic environmental conditions like rain in this study.

C. Navigation

The simulator allows users to load a number of different obstacles and test navigation algorithms controlling an ASV,

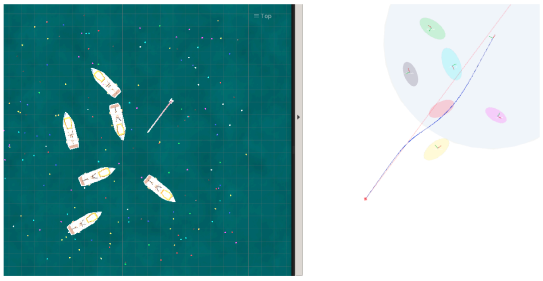

Fig. 10: Navigation scenario with a number of different boats representing traffic and controlled ASV. (left) view in Unity, (right) view from ROS rviz.

such as [31], [32]. Fig. 10 shows an example of such a scenario. Note that multiple ASVs can be controlled, so multi-ASV coordination can be tested as well.

VI. Conclusion and Future Work

This paper introduced CataBotSim, an open source ASV simulator implemented with Unity and ROS, and made multiple contributions. First, CataBotSim allows the user to drive an ASV in customizable environments and import custom or scanned physical terrain. This is highly beneficial for testing and validating machine learning models as it simplifies the creation of large diverse datasets without the cost of creating the same datasets in the physical world. Second, we have added sensors such as IMU, Cameras, GPS, LiDAR, and Marine Radar to CataBotSim, and we have added noise and weather effects to the sensor models to increase accuracy compared to physical sensors. To support training of machine learning models, the user can record sensor datasets in the csv file format or as images in the case of segmented, depth, normal, and color camera perception. Third, the user can simulate up to 100 additional boats by default (with the option to arbitrarily increase this limit), as well as different weather and physics components such as rain, wind zones, and buoyancy. All of these attributes can be activated and deactivated via a user friendly UI which can be configured at runtime or by a .json configuration file. Finally, CataBotSim is fully integrated with ROS, allowing external programs to read the current state of the simulation as well as control all vehicles.

In the near future, we are planning to introduce more weather conditions to our simulator. We will take advantage of more of Unity 3D's built-in features to improve the realism of the terrain visuals since it is currently using a gray checkerboard pattern. We will also be looking to see what other common communication and sensor hardware exist for ASVs that we can integrate into the simulator to increase the types of scenarios it can handle. We plan to add options for realistic decreased sensor accuracy such as material surface reflectivity to affect Marine Radar and LiDAR as well as sensor bias and drift.

References

[1] U. Nations, *Review of Maritime Transport 2021*. United Nations Conference on Trade and Development, 2022.

[2] M. Freese, S. Singh, F. Ozaki, and N. Matsuhira, "Virtual robot experimentation platform v-rep: a versatile 3d robot simulator," in *Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. Springer, 2010, pp. 51–62.

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on robot learning*. PMLR, 2017, pp. 1–16.

[4] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta *et al.*, "Lgsvl simulator: A high fidelity simulator for autonomous driving," in *IEEE International conference on intelligent transportation systems (ITSC)*, 2020, pp. 1–6.

[5] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[6] M. Paravisi, D. H. Santos, V. Jorge, G. Heck, L. M. Gonçalves, and A. Amory, "Unmanned surface vehicle simulator with realistic environmental disturbances," *Sensors*, vol. 19, no. 55, p. 1068, Jan. 2019.

[7] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ international conference on intelligent robots and systems (IROS)*, vol. 3, 2004, pp. 2149–2154.

[8] B. Bingham, C. Agüero, M. McCarrin, J. Klamo, J. Malia, K. Allen, T. Lum, M. Rawson, and R. Waqar, "Toward maritime robotic simulation in gazebo," in *OCEANS 2019 MTS/IEEE SEATTLE*, 2019, pp. 1–10.

[9] I. Loncar, J. Obradovic, N. Krasevac, L. Mandic, I. Kvasic, F. Ferreira, V. Slosic, D. Nad, and N. Miskovic, "MARUS - a marine robotics simulator," in *OCEANS 2022, Hampton Roads*, 2022, p. 1–7.

[10] P. Smith and M. Dunbabin, "High-fidelity autonomous surface vehicle simulator for the maritime robotx challenge," *IEEE Journal of Oceanic Engineering*, vol. 44, no. 2, p. 310–319, Apr. 2019.

[11] Unity3D, "Simulating robots with ROS and unity," https://unity.com/blog/engine-platform/robotics-simulation-is-easy-as-1-2-3, [Accessed 15-03-2024].

[12] 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.

[13] J. Collins, S. Chand, A. Vanderkop, and D. Howard, "A review of physics simulators for robotic applications," *IEEE Access*, vol. 9, pp. 51 416–51 431, 2021.

[14] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a robot simulator for research and education," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2007, pp. 1400–1405.

[15] UnrealEngine, "Unreal Engine — The most powerful real-time 3D creation tool — unrealengine.com," https://www.unrealengine.com/en-US, [Accessed 15-03-2024].

[16] Unity Technologies, "Unity Perception package," https://github.com/Unity-Technologies/com.unity.perception, 2020, [Accessed 15-03-2024].

[17] P. Namal Senarathne, W. S. Wijesoma, K. W. Lee, B. Kalyan, M. Moratuwage, N. M. Patrikalakis, and F. S. Hover, "Marinesim: Robot simulation for marine environments," in *OCEANS'10 IEEE SYDNEY*, 2010, pp. 1–5.

[18] E. Potokar, S. Ashford, M. Kaess, and J. G. Mangelson, "HoloOcean: An underwater robotics simulator," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2022, pp. 3040–3046.

[19] Unity Technologies, "Unity Perception package," https://docs.unity3d.com/Packages/com.unity.sensorsdk@2.0/manual/index.html, 2024.

[20] coppeliarobotics, "Robot simulator CoppeliaSim: create, compose, simulate, any robot - Coppelia Robotics — coppeliarobotics.com," https://www.coppeliarobotics.com/, [Accessed 14-03-2024].

[21] P. Chang and T. Padif, "Sim2real2sim: Bridging the gap between simulation and real-world in flexible object manipulation," in *IEEE International Conference on Robotic Computing (IRC)*, 2020, pp. 56–62.

[22] arcgis, "arcgis.com," https://www.arcgis.com/index.html, [Accessed 15-03-2024].

[23] U. Technologies, "The New Water System in Unity 2022 LTS and 2023.1 — Unity Blog — blog.unity.com," https://blog.unity.com/engine-platform/new-hdrp-water-system-in-2022-lts-and-2023-1, [Accessed 15-03-2024].

[24] Unity3D, "Rain Maker - 2D and 3D Rain Particle System for Unity — Environment — Unity Asset Store — assetstore.unity.com," https://assetstore.unity.com/packages/vfx/particles/environment/rain-maker-2d-and-3d-rain-particle-system-for-unity-34938, [Accessed 14-03-2024].

[25] O. Inc, "Ouster OS1 Lidar sensor," https://data.ouster.io/downloads/datasheets/datasheet-rev7-v3p1-os1.pdf, [Accessed 15-03-2024].

[26] J. P. Espineira, J. Robinson, J. Groenewald, P. H. Chan, and V. Donzella, "Realistic lidar with noise model for real-time testing of automated vehicles in a virtual environment," *IEEE Sensors Journal*, vol. 21, no. 8, pp. 9919–9926, 2021.

[27] C. Goodin, D. Carruth, M. Doude, and C. Hudson, "Predicting the influence of rain on LIDAR in ADAS," *Electronics*, vol. 8, p. 89, 01 2019.

[28] M. Berk, M. Dura, J. Rivero, O. Schubert, H.-M. Kroll, B. Buschardt, and D. Straub, "A stochastic physical simulation framework to quantify the effect of rainfall on automotive lidar," vol. 1, 04 2019.

[29] A. B. Moody, "American Practical Navigator, Pub. No. 9 of the United States Defense Mapping Agency Hydrographic Topographic Center, originally by Nathaniel Bowditch. Two volumes, 7 × 10 in (approx. 18 × 26 cm). Vol. I, 1977. Vol. II, 1975." *Journal of Navigation*, vol. 32, no. 3, p. 454–456, 1979.

[30] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics YOLOv8," 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[31] M. Jeong and A. Quattrini Li, "Motion attribute-based clustering and collision avoidance of multiple in-water obstacles by autonomous surface vehicle," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 6873–6880.

[32] Y. Kuwata, M. T. Wolf, D. Zarzhitsky, and T. L. Huntsberger, "Safe maritime autonomous navigation with colregs, using velocity obstacles," *IEEE J. Oceanic Eng.*, vol. 39, no. 1, pp. 110–119, 2014.