Development of an Underwater Simulator using Unity3D and Robot Operating System

Akash Chaudhary
Department of Physics,
Birla Institute of Technology and Science - Pilani
Goa, India
f20160743@goa.bits-pilani.ac.in

Rajat Mishra
Acoustic Research Laboratory,
National University of Singapore
Singapore
rajat@nus.edu.sg

Bharath Kalyan
Acoustic Research Laboratory,
National University of Singapore
Singapore
bharath@nus.edu.sg

Mandar Chitre
Acoustic Research Laboratory &
Department of Electrical and Computer Engineering,
National University of Singapore
Singapore
mandar@nus.edu.sg

***Abstract*—We developed an Open-Source Underwater Simulator that uses Unity3D for the simulating Remotely Operated Vehicle (ROV) operations and Robot Operating System (ROS) as the middleware to control the simulated vehicle. This simulator provides a virtual environment in which a vehicle can be simulated with a multi-beam SONAR and interact with features such as hilly terrain, lake, and pipelines. The vehicle is fitted with sensors and cameras to continuously monitor its state. The end-user is presented with an interactive user interface, which allows them to control various elements of the simulation. The application connects with ROS over a WebSocket to receive commands and send back parameters, video feed, and SONAR data, allowing the user to map the vehicle's environment in 3 dimensions using a mapping library. The simulator is tested with an adaptive lawn-mower type planner to display its full capabilities.**

***Index Terms*—underwater robotics, simulation, tether-less ROV inspection, Unity3D, Robot Operating System**

I. Introduction

Simulations serve as the backbone while developing algorithms for robots [10]. They allow researchers to test their system robustly in an virtual environment with relatively low risk and expenditure as compared to field experiments. This also provides the flexibility to test the boundary conditions and the system performance under different operating conditions. In general, many well-built simulators are available for aerial and ground vehicles such as AirSim [9], which allows the user to configure the vehicle controls and provides pre-built features to experiment. In the case of underwater missions, the challenges are different.

There are two major classes of vehicles used in underwater missions, Remotely Operated Vehicles (ROVs) and Autonomous Underwater vehicles (AUVs). ROVs are controlled by the operator to explore an intended region. On the contrary, AUVs can carry out a mission without the need for constant human supervision. The ROVs are generally tethered to a base station for remote control of the complete vehicle. If the tether can be replaced by a wireless link, we can increase the operating region of the ROV and reduce the risk of entanglement. While acoustic wireless links provide numerous advantages, they are much slower than tethered connections and exhibit long latency [11]. This results in a situation where we can have tether-less control over ROVs but with limited bandwidth, and sophisticated control algorithm.

We wanted to create a tool that can provide us with a platform to test and simulate algorithms under different operating conditions.

Over the years, many underwater simulators have been developed. UWSim [1] uses OpenSceneGraph (OSG) and osgOcean libraries for simulation. UWSim gives users the option to add various scene elements and supports multiple robots. It supports customizable widgets which can be placed on the main window to displays various data parameters as required. However, it has limited flexibility to create custom user interfaces and add various sensors like an IMU or a Doppler Velocity Log. UUV_Simulator [2] uses Gazebo for simulation along with Robot Operating System (ROS). In addition to its capability of simulating an underwater environment, it can also add new simulation scenarios at runtime, thus making it easier for the users to test their platforms. Even with the robustness of ROS and good features, several crucial components are missing. For example, the characteristically reduced visibility in a marine environment as we see further in the simulated ocean.

We are interested in mapping an area without any prior information of the environment. For this application, we required a setup that is capable of simulating an underwater environment and compatibility with ROS for easy integration with mapping packages. Therefore, simulators like the URSim [3], which are developed using Unity3D but do not contain mapping capabilities, were not a viable option for us. We were not able to find an existing solution that met our needs.This motivated us to develop a new simulator for tether-less ROV

operations.

We present an open-source underwater simulator developed using Unity3D and ROS. This simulator provides a good approximation of real-world ROV operations such as pipeline inspection and tether-less control.

The simulator is open-source in order to allow users to modify it according to their requirements. A comprehensive documentation makes it possible to quickly get acquainted with the development tools and Unity's user interface. The source code can be found at https://github.com/org-arl/UWRoboticsSimulator.

The simulator can take control commands from ROS and execute them in Unity while simulating the physical forces and constraints involved in the movement of an underwater vehicle. It also emulates a multi-beam SONAR allowing it to map its surroundings acoustically. In the following section we describe the reasons for selecting Unity3D as our physics engine, its capabilities, and its usefulness in our application. We present the details of the simulator's elements, including the environment building and the User-Interface. In Section IV, we explain the integration of ROS as the control center, thus making it possible to test any algorithm made in the ROS ecosystem to be tested in our simulator. The details about the emulation of the multi-beam SONAR are present in Section V and the simulator's mapping capabilities are discussed in Section VI along with the possibility of a digital twin control implementation. Finally, we discuss the results in Section VII.

II. Unity 3D

In general, Gazebo is the tool of choice for many researchers [2] when working with ROS and robotic systems. It is a helpful application due to its ability to create simulations that can be easily linked with ROS. Nevertheless, our project required more freedom and functionality that usually do not fall under Gazebo's domain, such as independent operation as an application and custom user interface. Unity3D is a powerful development platform that can simulate real-world interactions more accurately than Gazebo [3]. It can also connect with ROS over a ROSBridge WebSocket connection giving it the capability to be controlled from anywhere globally, provided that they are on the same network. Therefore, we used Unity3D as our simulation platform.

Unity3D is usually employed to develop games and is therefore quite adept at simulating ROV operations such as pipeline inspection. It has a well-configured physics engine, allowing us to simulate complex physical interactions easily. The engine can also simulate forces such as gravity, buoyancy, drag, and others. Unity can also be used to test vision-based algorithms. It allows us to control and set the visual elements of the environment, bringing it closer to what an actual camera would see in the same scenario. Unity also supports multiple platforms, so any application developed in Unity can be deployed on Windows, Linux, and macOS.

A. Terrain and Environment

Our test environment (Fig. 1) consists of a lake surrounded by mountains. The lake contains several objects such as pipelines and gates that can serve as metrics for performance testing of the simulator and its control as well as mapping capabilities. The scene also contains a control station that harbors the ROV. The seabed is a separate object used for texturing purposes and as a reference object for various functions involving depth information.

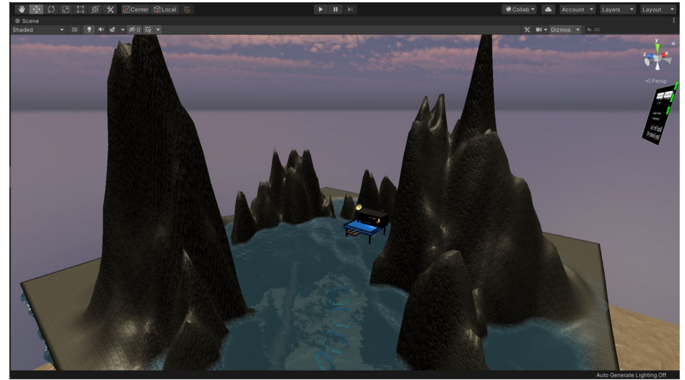

Fig. 1. The simulator environment containing the terrain, the vehicle and various scene components.

Fig. 2. The difference between the scene with and without underwater visual effects.

Along with functionality, realism also plays a vital role in the simulator. We created the seabed using simplex noise, and custom-built effects are used to simulate the underwater environment as shown in Fig. 2. A distortion effect is added to the camera view to depict the light refraction that occurs underwater, and a fog to resemble the reduced visibility in turbid waters. Light diffusion effects are also added to simulate sunlight diffusing through the water surface and falling on the underwater objects.

B. Vehicle Dynamics and Control

The vehicle designed in our simulator is based on a ROV with six degrees of freedom controlled using the appropriate thrusters. The ROV can translate along three directions and rotate about three axes. The corresponding movements are:

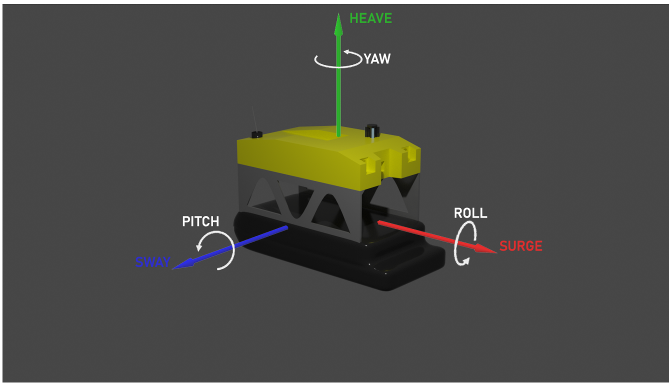

Fig. 3. The Remotely operated underwater vehicle along with its movement vectors.

- Translation:
  - Surge - Along x-axis
  - Sway - Along y-axis
  - Heave - Along z-axis
- Rotation:
  - Roll - About x-axis
  - Pitch - About y-axis
  - Yaw - About z-axis

Fig. 3 demonstrates the vehicle dynamics with the model used in the simulator. The AddForce mechanism present in Unity3D is used to emulate thrusters in which forces and torques are applied along the axis per the control input. This closely mimics the behavior of real thrusters. The thrusters are controlled by commands sent from ROS.

C. The Simulation Control

The simulation environment is controlled with a variety of scripts, each enabling crucial functionality. Some of the important script groups are :

1) *Buoyancy:*
   a) BoatPhysics: This script is attached to the vehicle and acts a parent for the other buoyancy scripts.
   b) ModifyBoatMesh: Generates the mesh that is below the water.
   c) TriangleData: Sends the triangle information to other Buoyancy scripts.
   d) WaterController: Extracts information about the water and water level to send to the other scripts.
2) *Button Functions:*
   a) Controls the toggle functions such as ROV release from the control station and light panels of the ROV.
   b) Ability to switch between camera display modes to focus on a specific camera view.
3) *Pause Game:* Encodes a pause menu to give more control over each simulation run.
4) *Seabed Script:*
   a) Calculates Vehicle's Angular and Linear Velocity and convert them to their respective scalars i.e. Angular and Linear Speed.
   b) Calculates the Acceleration of the Vehicle.
   c) Calculates the vehicle's distance from the seabed.
   d) Calculates the Vehicle's depth, its surrounding Temperature and Pressure.
   e) Calculates the Vehicle's distance from the control station.
   f) Controls the behavior of the Range warning, Proximity warning and the collision warning LEDs.
5) *StartScreen:* Script controlling the elements of the main menu including user input for IP address, port number, mission timing and SONAR Model. Takes these inputs and uses them in the simulator and redirects to the simulator screen when 'Start' is pressed.
6) *UIScript:* Creates and display various UI elements such as the vehicle's speeds, acceleration, position and various sensor outputs.

There is another set of scripts which are responsible for the transfer of data to and from the ROS system. They will be discussed separately in section IV.

D. Physical Forces and Interactions

One of the advantages of using Unity is the availability of rigidbody properties, which gives the vehicle its physical properties like mass and enables collision detection. Once the vehicle is given the rigidbody property, it can be influenced by gravity, and drag (and angular drag) forces can be applied to the vehicle. These drag forces and their proportional magnitude can be controlled from the inspection panel of Unity, giving us more control over the overall movement behavior of the ROV.

When working with an underwater environment, a prominent force exerted on the vehicle in water is the Buoyant force, which manifests due to the imbalance in the vertical component of the hydrostatic forces on the surface of a body. If the volume of the submerged body is known, it can be calculated using Archimedes' principle:

$$F_b = -\rho g V$$

where, F_b is the buoyant force, ρ is the fluid density, g is acceleration due to gravity, and V is the volume of the displaced fluid.

While this method works perfectly well in theory, game engines require us to discretize the problem, which means that we would need to divide the vehicle into primitive shapes to calculate the volume of the submerged part. It also requires us to close the submerged volume to calculate its value [7]. Even primitive shapes such as spheres, which reduce the required computational power, yield inefficient approximation of the volume of a body. This leads to a higher error in low polygon bodies. We, therefore, adopt the surface approximation approach proposed in [6] and [7].

We start by dividing the surface of our vehicle into triangles. A function then determines, for each triangle, whether it is

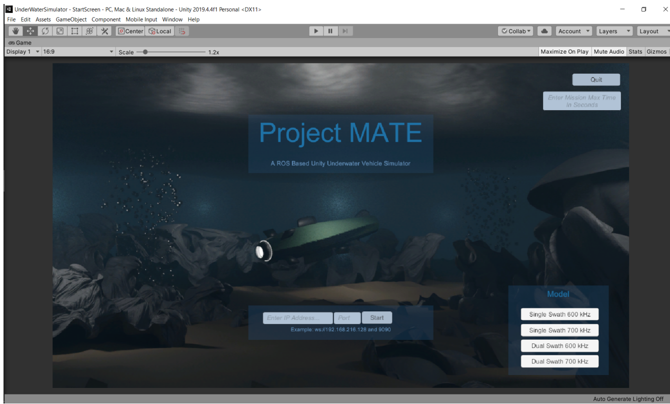

Fig. 4. The Start Screen allows the user to set the parameters of the simulation.

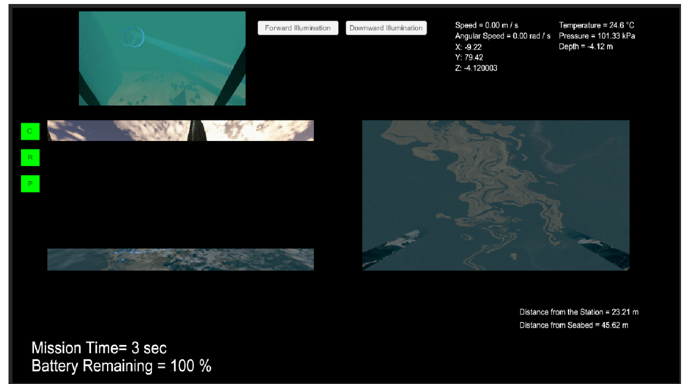

Fig. 5. The main user interface displaying the vehicle parameters, camera feeds and warning lights.

submerged or not (in the case of partially submerged ROV). The triangles are divided into three categories: Submerged triangles, not submerged triangles, and triangles intersected by the water surface. We discard the not submerged triangles from our calculations as they do not contribute to the buoyancy. The intersected triangles are further divided into smaller triangles. This is done till every triangle is either above or below the water surface. The buoyant force is then added to each submerged triangle, thus giving us the overall force on the vehicle.

This method removes any constraints on the vehicle shape, and any custom model can be used for the simulation. The computational cost is also low as game engines work well with triangular calculations. Introducing the buoyant force in the simulation also forces the user to compensate for it in their controller, just like they would do in a real ROV.

E. Sensors and Attachments

To make any control possible, the vehicle needs to have a base set of sensors that can localize the vehicle and record environment parameters as part of an exploration mission. The ROV included in the simulator contains a wide variety of sensors, and their output are communicated to the ROS system for further processing.

The simulated ROV has two cameras, one facing forward and the other facing downwards. We also emulate an Inertial Measurement Unit (IMU) to output angular and linear velocities and accelerations. It also has a proximity sensor for safety and pressure and temperature sensor to monitor the environment. The pressure sensor is also used to find the depth of the vehicle. In order to enable navigation in relatively dark environments, there are navigation lights on the vehicle. These can be toggled on and off using command inputs from ROS.

F. The User Interface

The simulator contains an intuitive User Interface (UI), allowing the user to get accustomed to the application quickly. The UI is similar to a small conventional ROV control station.

The UI is divided into two screens. The first one is the Start Screen (Fig. 4). As the simulator uses ROS for controlling the vehicle, the user can start a ROS server and connect it with the simulator using the ROSBridge. The simulator UI provides the choice of using four SONAR model variants depending upon the environment. Once the simulation is launched, the user is taken to the main screen as shown in Fig. 5. It shows various vehicle parameters, such as its estimated position, speed, angular speed, pressure, temperature, depth, and distance from the station. It also simulates LED lights that warn the user of any collision or possible communication problems due to the vehicle going out of range.

III. Blender

All the components of the simulation were modeled using Blender. *Blender* is an open-source 3D design tool for creating 3D models, environment design, animations, and game objects. This 3D design tool made it possible to make custom models for our simulator, therefore significantly increasing the customizable properties of the simulation. Objects are exported as Filmbox (.fbx) file type from blender to be imported into Unity3D. The objects might not carry over custom materials and shaders from Blender to Unity. They will also not have colliders attached to them on import, so special care was taken while using them in a scene. The major difference comes in how the built-in blender shaders behave in Unity, making it necessary to reassign them once imported.

IV. Robot Operating System

ROS [8] is an open-source framework that provides packages and tools to develop middleware for robots. It is one of the most widely used tools amongst robotics researchers and engineers. It provides us with all the required utilities to set up our system and build a middleware to control the simulated robot. The agents within ROS use a publisher - subscriber method to exchange both structured and unstructured data. In addition, it allows us to visualize the data if required, like in the case of stored maps.

We wanted to build a simulator that was compatible with ROS, as many robotics systems today use ROS as their framework. Our system consists of three separate parts: Control, Sensors and Mapping.

1) Control: To send commands from ROS to Unity, this section provides us with the tool to manages the control messages for the simulated vehicle in our unity3D simulator. It includes movement commands, camera control, lights, and the release of the ROV from the command station. It also sets up the interface for our planner to interact with the simulator.
2) Sensors: This part supervises the sensor outputs from the simulated vehicle, like pressure, temperature, position, and movement feedback, and publishes them for other programs to use. It also publishes the transform of the vehicle to be used by the mapping system.
3) Mapping: The last part is the mapping capabilities of our simulator. This part parses the information sent from the Unity system and uses it to create a map of the surroundings and store it for future use. This is explained in detail in sections V and VI.

We use the ROSBridge library to connect ROS with our Unity3D simulator.

A. *ROSBridge Library*

The ROSBridge library provides a JSON interface to ROS, allowing us to publish and subscribe to ROS topics, call ROS services and most importantly, use Web sockets. We use the ROSBridge library over a WebSocket, allowing our Unity system to connect to ROS and behave as an integrated system.

We send data, including parameters, control inputs, or video feed, between Unity3D and ROS, enabling us to use the Unity simulator as a substitute for actual hardware. This allows us to test all of our algorithms as if running them on the real ROV. This saves us time and money, and it also reduces risks that experimental programs might pose on expensive hardware. The ROSBridge for Unity is a C# based library that serves as the other end of this exchange. Michael Jenkin wrote the original library, but we are using a modified version written by Mathias Ciarlo [4].

We use ROSBridge to send video feeds, depth feed, and vehicle parameters from Unity to ROS and receive control commands from ROS. The setup uses the RosInitializer script to input the target IP address and calls all the subscriber and publisher scripts in Unity. It also projects the camera feed on to 2D textures to be transferred to ROS as a compressed image.

To ensure that our system resembles real hardware as closely as possible, we introduced an artificial lag in the communication pipeline to emulate the communication lag over an acoustic modem. The delay depends on the distance of the vehicle from the command station to account for the travel time of sound.

We tested our system with ROS Melodic and Noetic, the latest version of ROS on publication date. As ROS is approaching its end of life, to ensure our simulator's longevity, we also tested it with ROS2 Eloquent and Foxy.

V. SONAR

SONAR is a technique that uses sound propagation to navigate, and detect objects on or under the surface of the water. We emulated a sonar in our Unity3D vehicle to send depth information over our ROSBridge.

Fig. 6. Depth map created using shader modification

A. *Depth Maps*

The first method we tested was based on depth cameras like the Intel Realsense or Zed camera, in which the system outputs a depth map, with its pixel colours indicating the depth information [12] (Fig. 6). We achieved this by modifying the shaders of the camera output. We could control the resolution, range, and other parameters.

This approach required us to modify shaders at runtime, making it computationally expensive. To solve this, we tested a method provided by Unity known as Raycast, which closely resembles a SONAR.

B. *Unity3D Raycast*

Raycast is a feature of Unity, where a ray propagates from an object and returns a Boolean based on whether it hit an object or not. It is usually used to detect bullet collisions in games, as displayed in Fig. 7. While the Raycast is helpful to detect collisions in the ray's path, it can also return the hit distance, and therefore can be used as a SONAR which returns the distance of the object it hits.

C. *Multi-beam SONAR*

Using multiple beams enables the user to cover a larger area and create a dense map of the environment. The first step towards building a Sonar in our simulator was to identify a real-world product to base our model. The EM 2040 MKII MULTIBEAM ECHOSOUNDER [13] (Fig. 8) was chosen as it fulfills our range as well as swath spread angle criteria and comes in multiple variants, giving the user more control depending upon the mission's environment. We implemented four variants of the sonar, 600Hz and 700 Hz, with the option for Single or Dual Swath, which will change the range and the spread angle of the equipment.

We have set-up the sonar in a forward-facing configuration. When the simulator is launched, the user is given the option to choose a model variant. A model trigger is sent from Unity to ROS as soon as the user clicks on the start button so

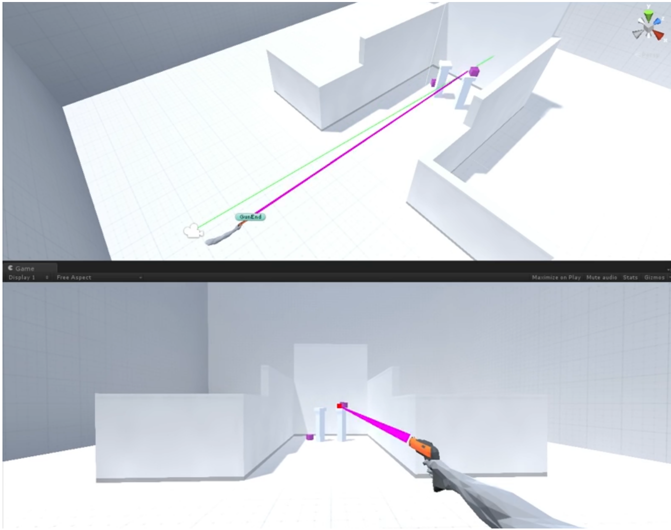

Fig. 7. Raycast being used in Unity to find bullet collision

Coverage example for EM 2040 with bottom type rock (BS = - 10 dB), NL = 45 dB, FM enabled

Operating mode	Cold ocean water			Cold fresh water		
EM 2040-04:	Max depth	Max coverage single RX	Max coverage dual RX	Max depth	Max coverage single RX	Max coverage dual RX
200 kHz	635 m	920 m	980 m	1360 m	1990 m	2110 m
300 kHz	480 m	670 m	760 m	740 m	1100 m	1270 m
400 kHz	315 m	410 m	430 m	430 m	570 m	610 m
600 kHz	95 m	130 m	-	115 m	150 m	-
700 kHz	55 m	27 m	-	60 m	30 m	-
EM 2040-07:						
200 kHz	600 m	880 m	930 m	1300 m	1870 m	2000 m
300 kHz	465 m	640 m	725 m	700 m	1050 m	1200 m
400 kHz	300 m	385 m	410 m	375 m	540 m	570 m
600 kHz	85 m	120 m	-	105 m	140 m	-
700 kHz	50 m	25 m	-	55 m	28 m	-

Fig. 8. The EM2040 MKII Multibeam Echosounder and its variants.

that the ROS system can know which variant of the SONAR is selected. The ROS system adjusts the related parameters like the range and swath angle accordingly. The output "hit" distance of every ray is stored in a single array which is then communicated to ROS over ROSBridge for further processing. The sonar can also tilt upwards or downwards, giving it the ability to scan vertical structures without moving the vehicle.

We can control the tilt angle of the sonar by sending control commands from ROS to Unity. They are added progressively to the movement of the scanner. The pose coordinates from the ROV to the scanner are sent to ROS, which are then converted into transforms by a tf2 broadcaster. This transforms as the base_scan frame for the laser scan topic and all for further processing. Fig. 9 shows the SONAR in action. Fig. 10 shows the transform tree.

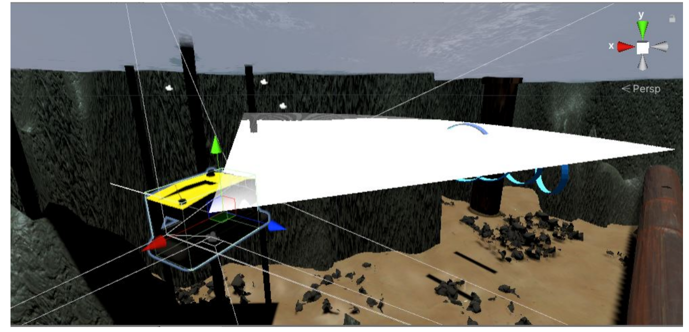

Fig. 9. The forward facing sonar using raycast method to return depth values.

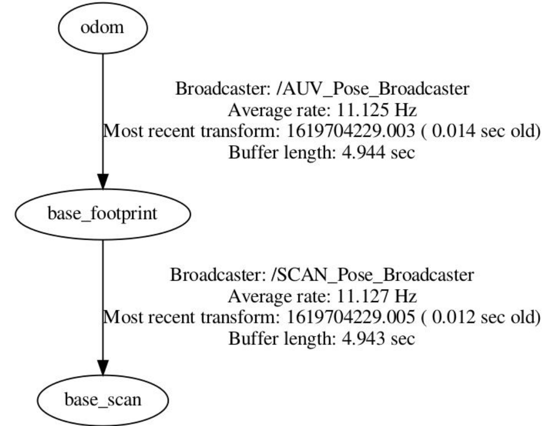

Fig. 10. Transform tree from the world frame to the SONAR sensor frame.

VI. Mapping with OctoMap

The OctoMap library [14] implements a 3D occupancy grid mapping approach, providing data structures and mapping algorithms in C++ particularly suited for robotics. We use OctoMap to recreate the environment scanned by our underwater vehicle and subsequently save it for future use. This is done in several steps, outlined below:

1) By incorporating the position published from the Unity3D simulator of both the vehicle and the scanner, we publish the respective transform by using the tf2 library. This is then broadcasted by a tf broadcaster to be used by our OctoMap node later on.
2) A string decoder subscribes to the decoded depth array sent from Unity3D and republish that data as a float array.
3) The LaserScan Publisher uses the decoded depth float array to convert it into a laser scan topic. It also uses the transform of the sonar as the scanner link.
4) The OctoMap mapping tool needs the depth data as a point cloud. So we convert the laser scan to point cloud and publish it over the topic /laserPointCloud
5) Once we have our point cloud and transforms ready, we feed them into our OctoMap mapping node, where we can use the rviz tool to visualize the recreated map.

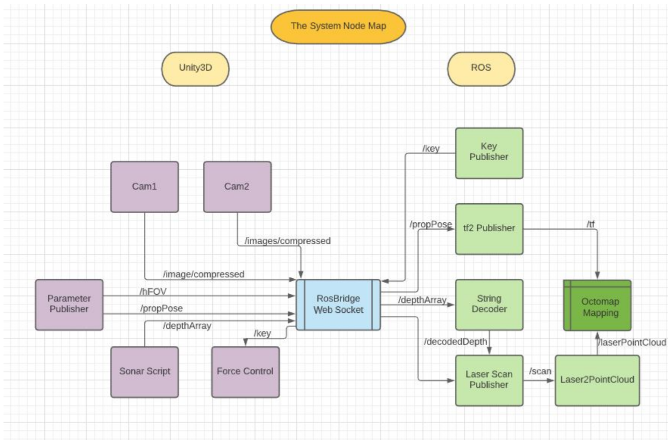

Fig. 11. System's nodemap displaying the mapping process.

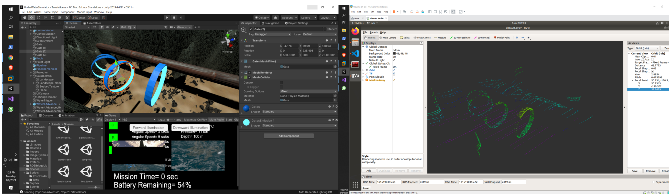

Fig. 12. Comparison between the terrain and the 3D occupancy grid map recreated in rviz.

6) The final Node Map of the whole system is represented by Fig. 11
7) A sample map is shown in Fig. 12

VII. Results and Discussion

The simulator is completely operational and is able to transfer data as well as camera feed over the ROSBridge connection. The control inputs from ROS are correctly executed in Unity3D. The physics engine simulates the interactions with the underwater objects as intended. The simulated SONAR output, along with the OctoMap package, is able to create a rich map, as shown in Fig. 12. The user interface gives sufficient control over the simulation and prepares the user for a real mission.

We tested the mapping pipeline for creating a "Digital twin" using lawnmower-like patterns. Control commands were sent from the ROS, and based on the feedback from the sensor, the patterns were auto-generated. Simultaneously, we mapped the surroundings using the SONAR and stored it in the system. This helped us simulate a real-world scenario where tether-less ROVs can be used for mapping underwater environments.

Our simulator can be used to develop a robust platform to design and test various planners and tools employed in marine robotics. It allows researchers to check their systems before deployment, saving time and money. It can be used for other applications such as human-in-the-loop-based control and as a training software for new ROV controllers. Furthermore, it can be used as an educational tool for students new to the field of marine robotics.

The simulator is being further enhanced. We intend to add more user-friendly control over the physical components of the simulator, making it possible to change simulation parameters and components even in the exported executable of the simulator. Meanwhile, the source code is freely available for anyone to use or modify it to meet their requirements.

Acknowledgment

This research is supported by NUS Advanced Robotics Centre Seed funding & A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Industry Alignment Fund – Pre-Positioning (IAF-PP) (Award A20H8a0241).

References

[1] Prats, M.; Perez, J.; Fernandez, J.J.; Sanz, P.J., "An open source tool for simulation and supervision of underwater intervention missions", 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2577-2582, 7-12 Oct. 2012

[2] M. M. M. Manhães, S. A. Scherer, M. Voss, L. R. Douat and T. Rauschenbach, "UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation," OCEANS 2016 MTS/IEEE Monterey, 2016, pp. 1-8, doi: 10.1109/OCEANS.2016.7761080.

[3] A. Konrad, "Simulation of Mobile Robots with Unity and ROS - A Case-Study and a Comparison with Gazebo," M.S. thesis, Dept. of Engineering Sc., Univ. West, Trollhättan, Sweden, 2019. Accessed on: July 29, 2020. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1334348/FULLTEXT01.pdf

[4] Mathias Ciarlo, *ROSBridgeLib*, GitHub, Jan. 6, 2015. Accessed on: Sept. 2, 2020. [Online]. Available: https://github.com/MathiasCiarlo/ROSBridgeLib

[5] Katara, Pushkal & Khanna, Mukul & Nagar, Harshit & Kumarappan, Annapurani. (2019). Open Source Simulator for Unmanned Underwater Vehicles using ROS and Unity3D. 1-7. 10.1109/UT.2019.8734309.

[6] Erik Nordeus, *Make a realistic boat in Unity with C#*, Habrador. Accessed on: Oct. 20, 2020. [Online]. Available: https://www.habrador.com/tutorials/unity-boat-tutorial/3-buoyancy/

[7] Jacques Kerner, *Water interaction model for boats in video games*, Gamasutra, Feb. 27, 2015. Accessed on: Oct. 20, 2020. [Online]. Available: https://www.gamasutra.com/view/news/237528/Water_interaction_model_for_boats_in_video_games.php

[8] Quigley, Morgan & Conley, Ken & Gerkey, Brian & Faust, Josh & Foote, Tully & Leibs, Jeremy & Wheeler, Rob & Ng, Andrew. (2009). ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software. 3.

[9] Shah S., Dey D., Lovett C., Kapoor A. (2018) AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In: Hutter M., Siegwart R. (eds) Field and Service Robotics. Springer Proceedings in Advanced Robotics, vol 5. Springer, Cham. https://doi.org/10.1007/978-3-319-67361-5_40

[10] Tselegkaridis, S.; Sapounidis, T. Simulators in Educational Robotics: A Review. Educ. Sci. 2021, 11, 11. https://doi.org/10.3390/educsci11010011

[11] Liu, Lanbo & Zhou, Shengli & Cui, Jun-Hong. (2008). Prospects and problems of wireless communication for underwater sensor networks. Wiley WCMC Special Issue on Underwater Sensor Networks. Wireless Communications and Mobile Computing. 8. 977-994. 10.1002/wcm.654.

[12] Unity Technologies, *Image Synthesis for Machine Learning*, Bitbucket, Dec 30, 2016. Accessed on: Oct. 5, 2020. [Online]. Available: https://bitbucket.org/Unity-Technologies/ml-imagesynthesis/src/master/

[13] KONGSBERG, *EM 2040 MKII Multibeam echosounder*. Accessed on: Dec. 15, 2020. [Online]. Available: https://www.kongsberg.com/contentassets/e8fa4f09f25f4b1e86eda52cc1355dc7/em-2040-mkii---data-sheet.pdf

[14] A. Hornung,. K.M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees" in Autonomous Robots, 2013; DOI: 10.1007/s10514-012-9321-0. Available: http://octomap.github.com