Coordinated Navigation and Localization of an Autonomous Underwater Vehicle
Using an Autonomous Surface Vehicle in the OpenUAV Simulation Framework

by

Harish Anand

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Masters of Science

Approved June 2020 by the
Graduate Supervisory Committee:

Jnaneshwar Das, Chair
Yezhou Yang
Spring Berman

ARIZONA STATE UNIVERSITY
August 2020
ABSTRACT

The need for incorporating game engines into robotics tools becomes increasingly crucial as their graphics continue to become more photorealistic. This thesis presents a simulation framework, referred to as OpenUAV, that addresses cloud simulation and photorealism challenges in academic and research goals. In this work, OpenUAV is used to create a simulation of an autonomous underwater vehicle (AUV) closely following a moving autonomous surface vehicle (ASV) in an underwater coral reef environment. It incorporates the Unity3D game engine and the robotics software Gazebo to take advantage of Unity3D’s perception and Gazebo’s physics simulation. The software is developed as a containerized solution that is deployable on cloud and on-premise systems.

This method of utilizing Gazebo’s physics and Unity3D perception is evaluated for a team of marine vehicles (an AUV and an ASV) in a coral reef environment. A coordinated navigation and localization module is presented that allows the AUV to follow the path of the ASV. A fiducial marker underneath the ASV facilitates pose estimation of the AUV, and the pose estimates are filtered using the known dynamical system model of both vehicles for better localization. This thesis also investigates different fiducial markers and their detection rates in this Unity3D underwater environment. The limitations and capabilities of this Unity3D perception and Gazebo physics approach are examined.
DEDICATION

This work is dedicated to my family, thesis advisors, and friends.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Jnaneshwar Das, Assistant Research Professor, School Of Earth and Space Exploration, for his immense support and guidance during the research, including technical discussions, asking fundamental questions, and the unconditional support.

Besides my advisor, I would like to thank my thesis committee members, Dr. Yezhou Yang, Assistant Professor, School of Computing, Informatics, and Decision System Engineering and Dr. Spring Berman, Associate Professor, School for Engineering of Matter, Transport and Energy, for their interest in the work and their insightful comments. I would like to thank Dr. Greg Asner and Dr. Robin Martin, scientists at Arizona State University’s Center for Global Discovery and Conservation Science (GDCS), for their inspiring efforts to conserve our coral reefs. I want to thank Dr. John Burns, Assistant Professor, University of Hawaii, whose 3D coral reef model is used in our simulation environment. I want to thank my labmates Alex, Cole, Devin, Luiza, Prasad, Rodney, Sarah, Zion, and Zhiang for their insights into the research work and the fun we had during our time at the lab. I want to thank Sarah for her huge support, the valuable suggestions and corrections she had for this document. I would also like to mention my roommates and friends - Avinash, Ashwin, Athira, Jacob, Sachin, Aswin, and Mamtha for the encouragement and support during the stressful and tiring days.
TABLE OF CONTENTS

| LIST OF TABLES                      | vi  |
| LIST OF FIGURES                    | vii |

CHAPTER

1 INTRODUCTION .......................... 1
   1.1 Motivation ................................ 1
   1.2 Thesis Statement ......................... 5
   1.3 Contributions and Assumptions .......... 5

2 UNDERWATER SIMULATIONS FOR LOCALIZING AN AUV WITH AN ASV .......................... 7
   2.1 Related Work ................................ 7
   2.2 Underwater Perception ..................... 8
       2.2.1 Light Attenuation in Water ............ 9
       2.2.2 Light Scattering ...................... 9
       2.2.3 Caustics ................................ 10
   2.3 Unity3D and Gazebo Communication .......... 11
   2.4 Unit Conversion ........................... 13
   2.5 Physics and Transformations ............... 14
   2.6 Camera Calibration ......................... 15
   2.7 Fiducial Markers ............................ 19
   2.8 Coordinated Navigation and Localization .. 22
   2.9 AUV Controller .............................. 26

3 THE OPENUAV SIMULATION FRAMEWORK .............. 29
   3.1 Related Work ............................. 30
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1 AirSim</td>
<td>30</td>
</tr>
<tr>
<td>3.1.2 FlightGoggles</td>
<td>31</td>
</tr>
<tr>
<td>3.1.3 Pavilion</td>
<td>32</td>
</tr>
<tr>
<td>3.2 System Design</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1 Simulation</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1.1 Gazebo</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1.2 Robot Operating System</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1.3 PX4</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1.4 QGroundControl</td>
<td>35</td>
</tr>
<tr>
<td>3.2.1.5 Unity3D Game Engine</td>
<td>35</td>
</tr>
<tr>
<td>3.2.2 Virtualization</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2.1 Containerization (Docker)</td>
<td>37</td>
</tr>
<tr>
<td>3.2.2.2 NGINX</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3 Interactive components</td>
<td>38</td>
</tr>
<tr>
<td>3.2.3.1 TurboVNC</td>
<td>39</td>
</tr>
<tr>
<td>3.2.3.2 NoVNC</td>
<td>39</td>
</tr>
<tr>
<td>3.2.3.3 Secure Shell Filesystem (SSHFS)</td>
<td>40</td>
</tr>
<tr>
<td>3.2.4 Academic Applications</td>
<td>40</td>
</tr>
<tr>
<td>4 LIMITATIONS AND FUTURE WORK</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>4.2 Simulation Limitations</td>
<td>42</td>
</tr>
<tr>
<td>4.3 Future Work</td>
<td>44</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>47</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Coordinate System Conversions between Unity3D and ROS</td>
<td>13</td>
</tr>
<tr>
<td>2. Comparison of OpenUAV-1 and the Improved OpenUAV</td>
<td>29</td>
</tr>
<tr>
<td>3. Evaluation of Publish Rate for 640x480 Images</td>
<td>43</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Global Airborne Observatory</td>
<td>2</td>
</tr>
<tr>
<td>2. AUV-ASV Methodology for Coral Reef Mapping</td>
<td>3</td>
</tr>
<tr>
<td>3. Attenuation of Light in Water at Various Depths</td>
<td>10</td>
</tr>
<tr>
<td>4. Unity3D Underwater Light Scattering in a Coral Reef Environment</td>
<td>11</td>
</tr>
<tr>
<td>5. Unity3D Underwater Caustics on the Coral Reefs</td>
<td>12</td>
</tr>
<tr>
<td>6. Unity3D and Gazebo Communication</td>
<td>13</td>
</tr>
<tr>
<td>7. Autonomous Underwater Vehicle, UDrone</td>
<td>15</td>
</tr>
<tr>
<td>8. The Coordinate Frames and Transformations for the ASV and AUV Simulation</td>
<td>16</td>
</tr>
<tr>
<td>9. Calibration of Unity3D Cameras in the Underwater Environment</td>
<td>17</td>
</tr>
<tr>
<td>10. Marker Detection Pipeline</td>
<td>19</td>
</tr>
<tr>
<td>11. Percentage of Detected Markers at Various AUV Depths</td>
<td>20</td>
</tr>
<tr>
<td>12. Detection of ArUco and AprilTag Markers</td>
<td>21</td>
</tr>
<tr>
<td>13. Illustration of the Visual (V) and Dynamical System (S) Model Fusion</td>
<td>23</td>
</tr>
<tr>
<td>15. Kalman Filter Estimates for the Position in Y-Direction</td>
<td>27</td>
</tr>
<tr>
<td>17. AUV-ASV Controller</td>
<td>28</td>
</tr>
<tr>
<td>18. OpenUAV Simulation Containers</td>
<td>30</td>
</tr>
<tr>
<td>19. Components of an OpenUAV Container</td>
<td>34</td>
</tr>
<tr>
<td>20. Unity3D Rendering of an Underwater Environment in OpenUAV</td>
<td>36</td>
</tr>
<tr>
<td>21. Unity3D Rendering of a Volcanic Plume in OpenUAV</td>
<td>37</td>
</tr>
<tr>
<td>22. OpenUAV System Architecture</td>
<td>40</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>23. OpenUAV Used as Part of SES 494/598 Autonomous Exploration Systems Course</td>
<td>41</td>
</tr>
<tr>
<td>24. A Swarm of AUVs Mapping a Coral Reef</td>
<td>45</td>
</tr>
</tbody>
</table>
1.1 Motivation

With warming waters, acidifying seas, destructive fishing, and anthropogenic impact of agricultural and industrial practices, our coral reef ecosystems face grave danger [Asner, Martin, and Mascaro 2017; Carlson, Foo, and Asner 2019]. To better understand the health of reefs, vast spatial scales need to be mapped, frequently. Asner and Martin, along with other scientists at the Arizona State University’s (ASU) Center for Global Discovery and Conservation Science (GDCS), are mapping reefs along the Hawaii islands at unprecedented spatial and temporal scales using a low flying airplane (Dornier 228-202) called the Global Airborne Observatory (GAO). The airplane is equipped with advanced sensors such as High Fidelity Visible-Shortwave Infrared (VSWIR) Imaging Spectrometer, waveform Light Detection and Ranging (wLiDAR) Scanner, High-resolution Visible-to-Near Infrared (VNIR) Imaging Spectrometer, and High-resolution Digital Imaging Camera [Asner et al. 2020; GDCS 2020; Tullis 2019]. Figure 1 shows the GDCS’s Global Airborne Observatory and the advanced sensors inside the vehicle.

The critical challenge to coral reefs imaging is the difficulty for satellite and airborne cameras to see through the seawater. The GDCS team solves this problem by developing various correction models for atmospheric effects, water column attenuation, and sun glint removal [Thompson et al. 2017; Asner et al. 2020; AllenCoralAtlas 2020]. To validate the mapping algorithms, they conduct field experiments that collect geo-
Figure 1. Global Airborne Observatory (left) is an airborne observatory with an advanced sensor suite that can perform a detailed analysis of Earth. The advanced sensors suite inside the vehicle is shown on the right. The images are used with permission from Dr. Greg Asner, director of ASU’s Center for Global Discovery and Conservation Science.

Referenced coral reef data. Typically during a field experiment, a diver or underwater vehicle tows a surface buoy with a GPS to record the position every 2 seconds [Atlas 2020]. Field experiments are risky and time-consuming process for the divers that can be automated through a symbiotic localization of AUV and ASV. Figure 2 shows the AUV, ASV, and the GAO collectively involved in coral reef mapping.

Accurate localization of an AUV in an underwater environment is challenging because electronic signals such as GPS and Wifi get attenuated by water. As a result, acoustic systems with known precise GPS locations, like ultra-short baseline (USBL) system and long-baseline (LBL) system, are often used for underwater positioning. These systems use the travel time of the acoustic signal and sound speed profile in the water to calculate the range and transmit angle. Such systems are costly, and the coverage area is limited to the beacon area. Therefore, in this thesis work, we explore the localization of an AUV using an ASV equipped with fiducial markers in a visually realistic simulation environment. Realistically simulating underwater vision is vitally important to this approach’s success, so that the cameras mounted on the
Figure 2. AUV-ASV methodology for coral reef mapping. The data gathered through Global Airborne Observatory is validated through an AUV-ASV system. The ASV has GPS and radio communication, while the AUV sensors include Inertial Measurement Unit (IMU) and multiple cameras. The fiducial marker underneath the ASV helps the AUV maintain a fixed distance from each other. The AUV’s primary task is to map the coral reefs while following the ASV.
AUVs simulate light attenuation, scattering, and color degradation to have underwater effects.

Robotics is a field with the constant interplay between hardware and software development. We leverage the current state of the art hardware and software technologies developed as part of drones and vertical take-off and landing (VTOL) aircraft, to develop the control stack for our underwater and surface vehicles. The ASV and AUV are controlled using PX4, a flight controller stack developed for aerial, ground, and sea vehicles [Meier, Honegger, and Pollefeys 2015].

An autonomy that does not work in the simulated world will not work in the physical world, and autonomy that does work in the physical world will work in the simulated world. [Bingham et al. 2019]

While developing our underwater vehicle, we adhered to the design principle articulated by Bingham et al. 2019 to have the maximum number of failures in simulations than in the physical world. We find that cloud-based solutions can enable extensive simulations involving hardware systems and algorithms, and these solutions are exceptionally relevant during times when remote work is necessary, such as during the COVID-19 pandemic. Towards this end, we present a software architecture that provides remote desktop accessibility through browsers for researchers and students. Hence, the researchers can access GPU-enabled machines without being physically present at the lab and do collaborative software development, like remote demonstrations.

In this simulation framework, we develop an aquatic coral reef environment that uses Unity3D’s perception and Gazebo’s physics [Koenig and Howard 2004]. We offer a new simulation environment for testing and developing autonomy for autonomous underwater vehicles (AUVs) to map coral reefs and other ocean ecosystems. We utilize the structure-from-motion three-dimensional model of the coral reef provided to us by
Dr. John Burns, Assistant Professor, University of Hawaii in our simulations [Burns and Delparte 2017; Burns et al. 2018].

1.2 Thesis Statement

A GPS-denied underwater vehicle (AUV) can be localized in coordination with an ASV and fiducial markers for coral reef mapping.

We validate the above proposition using an end to end visually realistic simulation framework that addresses underwater physics and perception challenges.

1.3 Contributions and Assumptions

We demonstrate coordinated navigation and localization of an AUV using an ASV in a combined Gazebo and Unity3D simulations. The ASV is equipped underneath with a fiducial marker of known dimensions. We localize the AUV by estimating the camera pose from the marker as both vehicles move through the coral reef. For more realistic underwater vision, the cameras show a Unity3D based ocean rendering software called Crest [Bowles et al. 2017], which can model properties like light interaction with clear and turbid water and generate Gerstner ocean waves.

We make an important assumption in our underwater simulation environment; Gazebo defines the dynamics of all vehicle models, and the camera’s perception uses the Unity3D environment. Hence, the modeled ocean waves in Unity3D do not have collision effects on the vehicle’s body. Furthermore, it results in no water excitation around the vehicles and wake pattern behind a moving ASV.

We also focus on the development of a robotics simulation ecosystem for long-term
student and research engagement. Our approach is to modify the existing open-source framework OpenUAV [Schmittle et al. 2018], to incorporate visual realistic game engine support in order to satisfy academic and research specific goals [Anand, Chen, and Das 2019]. To achieve long-term student and research engagement goals, we satisfy the following requirements:

1. Enable accessibility to remote Linux desktop environments with very low interactive latency.
2. Demonstrate realistic visual simulations by combining Gazebo and Unity3D environments.
3. Achieve collaborative code development among researchers and students without the use of screen sharing software.
4. Natively support simulation of PX4 based aerial, ground and sea vehicles and their corresponding mission planning and ground control software packages.
5. Provide an easy to use, software development environment with support for remote code execution.
6. Replicate actual resource constraints of the vehicles in the simulation by having similar memory and computational capacity constraints.
7. Provide automated data recovery mechanisms to avoid data loss.
Chapter 2

UNDERWATER SIMULATIONS FOR LOCALIZING AN AUV WITH AN ASV

2.1 Related Work

The development of cheap AUVs offers an exciting new technology for exploring and monitoring our coral reefs and ocean ecosystems. In the near future, we can envision using a heterogeneous swarm of AUVs and autonomous surface vehicles (ASVs) to do coordinated exploration and mapping of marine biological systems [Manderson et al. 2017; Manderson et al. 2016]. However, there are still a few obstacles that need to be addressed before this idea can become a reality.

One such obstacle is underwater perception. Underwater perception remains a challenge because cameras on the vehicle experience light attenuation, scattering, reflections, and reef caustics. Therefore, in order to develop robust autonomy, we must be able to model underwater perception. We do this by utilizing OpenUAV’s Unity3D perception pipeline and Gazebo to demonstrate vision-based autonomy for an AUV in an underwater coral reef environment. A primary design criterion of the simulation software is to support pre-existing perception packages developed in ROS with minimal modifications. Such packages include marine simulators, like Unmanned Underwater Vehicle Simulator (UUVSim) [Manhães et al. 2016], which address challenges in underwater and ocean physics.

There are works in underwater localization involving acoustic modems, computer vision, and sonar-visual inertial systems [Fallon, Papadopoulos, and Leonard 2010; Bahr, Leonard, and Fallon 2009; Manzanilla et al. 2019; Carreras et al. 2003; Rahman,
Li, and Rekleitis 2018]. Some works also addressed fiducial marker-based localization, where the markers are used as landmarks on reefs or placed on pipelines [Jung, Li, et al. 2017; Jung, Lee, et al. 2017]. We approach AUV localization by attaching a fiducial marker of known dimensions to the underside of an ASV. We explore the detection rates and accuracy of the common fiducials in our simulation environment at various depths, and use our findings to explore improvements to the markers to increase the number of detections.

We then attempt to have the AUV follow the ASV as it traverses the coral reef. Our problem of coordinated navigation and localization of an AUV with an ASV is similar to that of the autonomous landing of an Unmanned Aerial Vehicle (UAV) on a moving fiducial platform. In the UAV problem, we attempt to land on a moving marker platform by estimating the relative pose of the UAV with respect to the platform [Araar, Aouf, and Vitanov 2017]. Our problem is similar, as we are estimating the relative pose of the AUV with respect to the marker on the moving ASV while attempting to follow the ASV.

2.2 Underwater Perception

The underwater perception is a challenging scenario because light undergoes refractions at various mediums such as glass, water, and air. Hence we do camera calibration in the underwater environment to correct for the distortions that can occur. However, this approach cannot model all possible refraction scenarios; for example, warm water currents during a deployment can result in an error in our estimations. We describe in the following sections, the significant perception improvements that are required for the underwater simulations.
2.2.1 Light Attenuation in Water

In nature, we find that the intensity of light decreases with depth in water. Along with this decrease in intensity, we find that wavelengths within the spectrum attenuate at different rates. For example, blue light gets absorbed the least relative to the other colors in the spectrum [Mascarenhas and Keck 2018]. In the Unity3D Crest ocean configuration, they refer to this absorption at different depths as fog effect on the objects. The assumption here is that the underwater light absorption behaves very similar to fog [Bowles et al. 2017; Flick 2020]. Although fog is a sparse approximation to the underlying natural light absorption process, it is a quick and reliable method to control the color and depth in the underwater camera renderings. In Figure 3, we illustrate light attenuation achieved in Unity3D through Crest framework at 15, 5 and 2 meters down, and also shows surface reflection at 0.1 meters above the water.

2.2.2 Light Scattering

Scattering is a primary light interaction that happens when light passes through a medium. It occurs when light rays collide with suspended particles in the medium, resulting in a change in their direction. The two main types of scattering are in-scattering and out-scattering. In-scattering happens when other light rays enter the path of the examined light beam. Out-scattering occurs when the light rays are deflected to different directions from the sampled light beam. Like attenuation, out-scattering decreases the intensity of the light ray [Pharr, Jakob, and Humphreys 2016]. Unity3D Crest environments do not include in-scattering in simulations, but they do model out-scattering by darkening the environment lighting for objects farther
Figure 3. Attenuation of Light in Water at Various Depths

away. This is done by adding a linear “fog effect” to the objects. A linear fog refers to a linear interpolation between the object’s color and the specified fog color based on the location of the object. An example of this is shown in Figure 4.

2.2.3 Caustics

Underwater caustics are caused by the concentration of light rays with different paths. These paths begin at the ocean surface. To create a visually realistic caustic
environment, an underwater caustic texture is applied to the coral reef surface as shown in Figure 5.

2.3 Unity3D and Gazebo Communication

Our visual realistic simulation experiment required writing ROS packages for model pose publishers, controllers, and image publishers. These packages rely on transferring information between Unity3D and Gazebo environments, so we utilized the ROS WebSocket based communication package referred as ROS-Sharp (Figure 7). For our experiment, we created the AUV and ASV models in the Gazebo URDF/SDF format and used the ROS-Sharp URDF to Unity3D game object script to load them into the Unity3D environment. We then wrote a script to publish the pose of the
Gazebo vehicle models to the Unity3D environment. The ASV and AUV game objects subscribe to these model poses, and use the information to update their positions and orientations. The Unity3D cameras on the ASV/AUV game objects then publish captured images as compressed images through ROS-Sharp, and the controller code uses these images to control the vehicle in Gazebo. It is important to note that the controller and perception packages developed can work with the PX4 and Gazebo environments regardless of whether or not the user has set up Unity3D configuration.

An added benefit to using ROS-Sharp is that it provides the necessary ROS-Unity3D coordinate system transformations. The Unity3D environment has a Left-Handed coordinate system (y-up world), while ROS uses a Right-Handed coordinate system (z-up world), as shown in Table 1.
Figure 6. Unity3D and Gazebo communication. This figure shows the communication loop between Gazebo and Unity3D. The Model Pose Publisher script publishes the pose of the vehicles to ROS, and Unity3D subscribes to these pose messages using ROS-Sharp. The cameras in Unity3D publish their images through Image Publisher C script, which are then subscribed by the controller script.

<table>
<thead>
<tr>
<th></th>
<th>Unity3D</th>
<th>ROS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Z</td>
<td>X</td>
</tr>
<tr>
<td>Right</td>
<td>X</td>
<td>-Y</td>
</tr>
<tr>
<td>Up</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Rotation</td>
<td>clockwise</td>
<td>counter-clockwise</td>
</tr>
</tbody>
</table>

Table 1. Coordinate System Conversions between Unity3D and ROS

2.4 Unit Conversion

In Unity3D, the scale of game objects and their positions have units referred to as game units [Goldstone 2009]. Game units do not have a fixed conversion to metric or imperial units. Game developers usually keep a fixed reference scale throughout game development to make sure all of the objects in the scene look
proportional. This approach will cause a problem for us as there is no direct metric measurement of the markers or vehicle size. We solve this problem by assuming the metric convention throughout the scene development. Additionally, the Gazebo model states are published in metric units. Therefore, we incorporate a measurement script developed for VR/AR applications to obtain the dimensions of game objects in metric units [VRChewal 2015].

2.5 Physics and Transformations

Our underwater vehicle named UDrone follows a similar vehicle design as that of the UUV Hippocampus in the PX4 flight stack [Duecker et al. 2018]. The significant change in the vehicle design from Hippocampus includes using a larger enclosure of dimensions 0.322m length and radius 0.108m and placing the four rotors at the backside of the vehicle as shown in Figure 7. In simulations, we use the UUV simulator’s underwater plugin to simulate the vehicle’s drag forces, and the four rotors on the vehicle are controlled using the Gazebo’s rotor plugin. We utilize the PX4’s rate controllers to achieve the desired thrust and orientation for the underwater vehicle. A Proportional Integral and Derivative (PID) controller was written over the rate controller to keep the AUV strictly follow the ASV.

There are two cameras on the vehicle, the first camera aims up towards the ASV, and the second camera faces forward, i.e., towards the coral reef. Figure 8 shows the 4 major reference frames present in the simulation. The transformation between different frames is also shown in Figure 8.
2.6 Camera Calibration

Typically, we do not need to calibrate cameras in simulated environments because we can define the intrinsic parameters and camera model. However, we cannot directly specify the camera's intrinsic parameters in Unity3D's physical camera model, we can only specify the field of view and the sensor width and height. For our simulation experiment, we defined the cameras with a 60-degree field of view and a square sensor size of 50 millimeters by 50 millimeters. These parameters can be modified to reflect a real camera.

In Figure 9, we show some of the images captured during the underwater calibration of Unity3D camera using the camera calibration software from the ros-perception package. The size of the square markers on the board was 3.88 centimeters. We rotated and translated the calibration board game object across the camera’s field of view while the software performed camera calibrations.

The following equation represents the projection of a point in 3D space to a camera
Figure 8. The coordinate frames and transformations for the ASV and AUV simulation. The top image shows the four major coordinates frames defined in our experiments. The world and marker frame is attached to the ASV while the camera and udrone frame is on the AUV. The bottom image shows the parent child relationships defined using ROS transformation package. The transformation between marker and camera is generated from fiducial markers while the rest are defined using static transformations.
Figure 9. Calibration of Unity3D cameras in the underwater environment. A sample of 12 images captured by ROS camera calibration software. The calibration board is rotated and translated across the field of view of the camera.
image plane,
\[ \mathbf{x} = K[R|T]\mathbf{X} \tag{2.1} \]
where \( \mathbf{X} \) and \( \mathbf{x} \) denote the homogeneous world coordinates and camera coordinates respectively. The \( R \) and \( T \) matrices represent the transformation from the world coordinate system to the camera coordinate system. The camera intrinsic matrix \( K \) is defined as,
\[
K = \begin{bmatrix}
\alpha_x & s & x_0 \\
0 & \alpha_y & y_0 \\
0 & 0 & 1
\end{bmatrix} \tag{2.2}
\]
where
- \( \alpha_x \) denotes a scale factor conversion to pixel coordinates in \( x \)-direction,
- \( \alpha_y \) denotes a scale factor conversion to pixel coordinates in \( y \)-direction,
- \( s \) is the skew,
- \( x_0, y_0 \) are the coordinates of the principal point.

From the calibration experiments, the calculated intrinsic parameters are as follows,
\[
K = \begin{bmatrix}
554.807617 & 0.000000 & 319.301545 \\
0.000000 & 554.638062 & 239.407675 \\
0.000000 & 0.000000 & 1.000000
\end{bmatrix} \tag{2.3}
\]

We use these calculated calibration values to generate the required camera info and rectified image topics. The rectified images are necessary for estimating the pose of the AUV from ASV using the fiducial markers.
2.7 Fiducial Markers

Fiducial markers are artificial markers of known size that are placed in a camera’s field of view so that they appear in the image produced. The dimensions of the fiducials along with their coordinates in the image plane can be used to estimate the pose of the calibrated camera. This problem of estimating the pose of a calibrated camera using n number of known 3D points and their corresponding 2D projections is a well-studied research problem referred to as the Perspective-n-Point (PnP) problem [Fischler and Bolles 1981]. In this problem, we are calculating the $R$ and $T$ matrices defined above from the n number of point correspondences.

As mentioned in the Unit conversion section, game developers usually keep a fixed reference for scale. We found that an added advantage to using fiducial markers is that they provide a fixed scale for all other objects in the scene.

We have considered two fiducials in our underwater environment: ArUco markers and AprilTag markers [Munoz-Salinas 2012; Olson 2011]. ArUco and AprilTag markers follow a generic marker detection pipeline that is described in Figure 10. The image initially undergoes an adaptive thresholding process where the threshold is calculated from the surrounding pixels. The contours on the thresholded image are evaluated to filter out non-polygonal and small contours, and then the candidate polygons are un-warped and the marker code is extracted. The obtained code is then compared to
a known set of codes to remove the incorrect marker detections. Correct markers are used to calculate the relative pose of the camera.

The ArUco markers compute the threshold as an average of the neighboring pixels. AprilTags follow a similar adaptive thresholding procedure, where the threshold is the average of the lowest and highest neighboring pixel intensities. We found that increasing the window size of the threshold can reduce the disconnections on the marker borders for markers placed on the reef’s surface. Other underwater field experiments have also compared the detection rates over an increasing window size marker threshold [Čejka et al. 2019]. Our findings align with their results.

![Figure 11. Percentage of detected markers at various AUV depths. This plot shows the percentage of detected markers vs the depth of the AUV.](image)

We evaluated the performance of both ArUco and AprilTag markers in a poor visibility condition in our simulation environment. The ArUco and AprilTag markers
Figure 12. Detection of ArUco and AprilTag markers. This Figure shows ArUco and AprilTag marker detection in the AUV’s upward camera at depths of 3 meters, 5 meters and 8 meters.

on the ASV are 35-centimeters x 35-centimeters (black border), on a 40-centimeter x 40-centimeter board. For this analysis, we measured marker detection time and the number of detected markers per 1000 frames from the AUV’s upward camera. As shown in Figure 16, the AprilTag markers are detected when the vehicle is up
to 16 meters deep while the ArUco markers are only detected up to 8 meters below. Even when ArUco markers lost detections beyond 8 meters depth, AprilTags still had close to 100% detections until 12 meters. The average detection time for ArUco and AprilTag markers was found to be seven milliseconds and 15 milliseconds, respectively. Figure 17 shows a collection of ArUco and AprilTag marker detection images at various depths. The marker detection rate is also a function of the size of the marker size.

2.8 Coordinated Navigation and Localization

To achieve coordinated navigation between the ASV and AUV, we need to integrate a dynamical system model along with the marker detection data. For simplicity, we assume that the motion of the ASV follows a straight-line path, and the AUV maintains a fixed $z$ distance of 5 meters from the ASV. We also assume that the world frame of the AUV is attached to the marker underneath the ASV, so the states of the AUV are represented with respect to this world frame, as mentioned in the Physics and Transformations section. For our task, we need to incorporate the motion of the ASV along with the states of the AUV. The velocity components of the ASV are considered as part of the states. Figure 13 represents the state estimation module where the developed system (S) model predictions occur at 100 Hz, and the vision (V) based measurement updates happen at 7 Hz.

We define the state and observation vectors of our AUV-ASV system as the following:

$$x = \begin{bmatrix} u_x & u_x & u_y & u_y & u_z & u_z & u_{v_x} & u_{v_y} \end{bmatrix}^T \quad (2.4)$$

$$z = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^T \quad (2.5)$$
Figure 13. Illustration of the visual (V) and dynamical system (S) model fusion. The V refers to the visual pose observations from fiducial markers, and S refers to the dynamical system model, represented as Kalman filter evolved over time $t$.

where

- $w_x$ denotes the AUV’s x position with respect to the world frame,
- $w_x$ denotes the AUV’s x velocity with respect to the world frame,
- $w_y$ denotes the AUV’s y position with respect to the world frame,
- $w_y$ denotes the AUV’s y velocity with respect to the world frame,
- $w_z$ denotes the AUV’s z position with respect to the world frame,
- $w_z$ denotes the AUV’s z velocity with respect to the world frame,
- $w_{v_x}$ and $w_{v_y}$ denote the ASV’s x and y velocity with respect to the world frame,
The state space representation of the modelled system is as follows:

\[
\begin{align*}
w_x^{(k)} &= w_x^{(k-1)} + \Delta t (w \dot{x}^{(k-1)}) - \Delta t (w \dot{y}^{(k-1)}) \\
\dot{w}_x^{(k)} &= \dot{w}_x^{(k-1)} \\

w_y^{(k)} &= w_y^{(k-1)} + \Delta t (w \dot{y}^{(k-1)}) - \Delta t (w \dot{y}^{(k-1)}) \\
\dot{w}_y^{(k)} &= \dot{w}_y^{(k-1)} \\

w_z^{(k)} &= w_z^{(k-1)} + \Delta t (w \dot{z}^{(k-1)}) \\
\dot{w}_z^{(k)} &= \dot{w}_z^{(k-1)} \\

w_{\nu}^{(k)} &= w_{\nu}^{(k-1)} \\
\dot{w}_{\nu}^{(k)} &= \dot{w}_{\nu}^{(k-1)}
\end{align*}
\]  

(2.6)

We describe a Kalman filter evolved at time \( k \) from state \( k-1 \) as follows:

\[
\begin{align*}
\dot{x}_k &= F \dot{x}_{k-1} + B u_k + w_k \\
\dot{z}_k &= H \dot{x}_k + r_k
\end{align*}
\]  

(2.7) (2.8)

where \( F \) is the state transition matrix, \( B \) is the control matrix, \( H \) is the measurement model matrix, \( w \) represents process noise, and \( r \) is the observation noise [Simon 2006; Labbe 2014]. The process noise is assumed to be a zero mean Gaussian noise with covariance \( Q \),

\[
w_k \sim \mathcal{N}(0, Q_k)
\]  

(2.9)

The observation noise is also assumed to be a zero mean Gaussian noise with covariance \( R \),

\[
r_k \sim \mathcal{N}(0, R_k)
\]  

(2.10)

We do not have a control input or control matrix in our filtering process, so the updated filtering steps become:
Predict step

\[ \bar{x}_k = F x_{k-1} \]  
\[ \bar{P}_k = F P_{k-1} F^T + Q_{k-1} \]  

(2.11)

Update step

\[ y_k = z_k - H \bar{x}_k \]  
\[ K_k = \bar{P}_k H^T (H \bar{P}_k H^T + R)^{-1} \]  
\[ x_k = \bar{x}_k + K_k y_k \]  
\[ P_k = (I - K_k H) \bar{P}_k \]  

(2.12)

where, \( \bar{x}_k \) and \( \bar{P}_k \) are the estimated prior mean and covariance of the states, \( y_k \) is the measurement residual or innovation term, \( K_k \) is the Kalman gain, and \( x_k \) and \( P_k \) are the estimated posterior mean and covariance of the states.

The state transition matrix \( F \) and observation matrix \( H \) for our system are as follows:

\[
F = \begin{bmatrix}
1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 & -\Delta t & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & -\Delta t & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]  

(2.13)

\[
H = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  

(2.14)

The Kalman filter estimates for \( x, y, z \) direction are shown in Figure 14, 15 and 16.
Figure 14. Kalman filter estimates for the position in x-direction. The orange line represents the ground truth obtained through Gazebo model states, green dots represent the ArUco x position observation with respect to the world frame and the blue line represents the Kalman filter estimation.

2.9 AUV Controller

We defined a simple Proportional Integral and Derivative (PID) controller for the AUV based on the Kalman filter estimates. The underwater vehicle has a PX4 based attitude controller as shown in Figure 17. We used 3 PID controllers on the AUV to control the relative position along x, y, z-direction. The first PID controller tries to achieve zero error on x-direction by adjusting the thrust on the vehicle. The second PID controller achieves zero error on y-direction by adjusting the yaw angle on the vehicle. The third PID controller attempts to maintain a 5 meter fixed distance on z-direction by adjusting the vehicle’s pitch angle.
Figure 15. Kalman filter estimates for the position in y-direction. The orange line represents the ground truth obtained through the Gazebo model states, green dots represent the ArUco y position observations with respect to the world frame, and the blue line represents the Kalman filter estimation.

Figure 16. Kalman filter estimates for the position in the z-direction. The orange line represents the ground truth obtained through the Gazebo model states, green dots represent the ArUco z position observation with respect to the world frame, and the blue line represents the Kalman filter estimation.
Figure 17. AUV-ASV controller. The ASV-AUV controller is developed over PX4’s AUV attitude controller where the input commands are thrust and orientation.
Chapter 3

THE OPENUAV SIMULATION FRAMEWORK

The OpenUAV simulation framework was initially developed by Schmittle et al. 2018. For a comparison of previous work with the improvements, we shall refer the initial work as OpenUAV-1. In order to achieve broader educational and research uses, we incorporated the system design from the works of Will Kessler and the Udacity team, and later, improved upon that to support photorealism and PX4 flight stack [Kessler 2018]. Table 2 shows a comparison of our contribution to the previous OpenUAV-1.

<table>
<thead>
<tr>
<th>Software</th>
<th>OpenUAV-1</th>
<th>Improved OpenUAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface</td>
<td>GZWEB, a Gazebo web interface</td>
<td>Lubuntu and noVNC</td>
</tr>
<tr>
<td>Photo-realism</td>
<td>No</td>
<td>ROS-Sharp and Unity3D</td>
</tr>
<tr>
<td>Remote Software development</td>
<td>No</td>
<td>Using SSHFS, a directory is shared between user and cloud simulation</td>
</tr>
<tr>
<td>Ground Control Support</td>
<td>No</td>
<td>QGroundControl</td>
</tr>
</tbody>
</table>

Table 2. Comparison of OpenUAV-1 and the improved OpenUAV

In Figure 18, a collection of Gazebo simulations are running simultaneously on the improved OpenUAV architecture, all accessed via a web browser.
A vibrant ecosystem of simulation tools exists for hardware and software development of PX4 based vehicles. We have focused on the simulation environment that supports PX4 based vehicles.

### 3.1.1 AirSim

AirSim is an open-source, cross-platform simulator built on Unreal Engine (UE4) that offers similar visually realistic simulations for drones and cars [Shah et al. 2018]. It supports hardware-in-the-loop simulations with flight controllers like PX4 and is compatible with the Robot Operating System (ROS) message passing interface [Quigley et al. 2009]. UE4 provides great benefits like access photo-realistic materials (Quixel), fast-texture rendering engines, and NVIDIA’s physics engine (PhysX).

With the lack of generic support for Unified Robot Description Format (URDF) and Software Definition Format (SDF) to create robot models and world physics, we find that UE4 robot models as game actors and working with them poses a
steep learning curve for students. Moreover, unlike Gazebo, AirSim requires further
code alterations when transitioning from the simulation environment to real-world
deployment. Hence, we focused our remote desktop ecosystem on a ROS-Gazebo-PX4
stack over an AirSim’s application programming interfaces (APIs). We utilize the
ROS-Sharp plugin to convert robot models specified in SDF and URDF format to
Unity3D game objects in our architecture [Bischoff 2017]. Through the SDF format,
we converted robot models like PX4’s iris drone, Turtlebot, heron, and a custom
underwater vehicle (UDrone) to Unity3D game objects [Garage 2011; Clearpath 2017].

3.1.2 FlightGoggles

FlightGoggles is a simulator that renders a virtual-reality environment around aerial
vehicles [Guerra et al. 2019]. In FlightGoggles, the sensor measurements are artificially
rendered in real-time while the vehicle vibrations and unsteady aerodynamics are
from the vehicle’s natural interactions.

As mentioned by [Guerra et al. 2019], photorealism is an influential component for
developing any autonomous flight controllers for outdoor environments. FlightGoggles
utilizes a Unity3D game engine to generate a real-time photorealistic rendering of
the environment. The exceptional advantage of the FlightGoggles framework is the
combination of real physics with the Unity3D-based rendering of the camera sensor’s
environment. Our goal of enabling the ROS-Gazebo-PX4 simulation stack with
Unity3D is to attain this essential photorealism component to outdoor simulations. In
this thesis, we demonstrate simulations involving a challenging outdoor scenario such
as coral reefs in an underwater environment. In future work, we address a possible
approach to attaining hardware in the loop simulations while the camera sensors display the Unity3D world.

3.1.3 Pavilion

Pavilion is a simulation environment that has both robotic perception and control developed in UE4 and ROS [Jiang and Hao 2019]. Pavilion provides an easy to use SDF to UE4 robot actors converter. In addition to SDF support, the authors use cROS, a the C-based ROS client to communicate UE4 actors with rosmaster process. Pavilion demonstrates a tightly-coupled ROS subsystem in UE4, and solves some of the code incompatibility issues often encountered in a UE4-based simulation systems. The authors address photorealism challenges in current robotics simulators by utilizing UE4’s physics and graphics. We find that our approach of passing kinematics from Gazebo to Unity3D can be applied to Pavilion as well, by utilizing cROS and SDF converter. However, Pavilion does not address the necessary code updates needed for a transition to real robotic systems. Hence, we attempt to create a software simulation ecosystem that supports faster outdoor testing, has a smooth learning curve for students, and has native support for photorealistic outdoor environment simulations.

3.2 System Design

The software components are classified into three categories: simulation, virtualization, and interactive components.
3.2.1 Simulation

Simulation has become a necessity to solve real-world problems safely and efficiently. Figure 19 shows an overview of a single OpenUAV simulation container, consisting of simulation and interactive components. The simulation components consist of the following packages.

3.2.1.1 Gazebo

The Gazebo simulator is a widely used open-source robotics simulator for designing robots, environments, and performing realistic rigid body dynamics [Koenig and Howard 2004]. The simulated objects have mass, velocity, friction, and other physical properties that enable simulating realistic physical simulations.

3.2.1.2 Robot Operating System

The Robot Operating System (ROS) is a robotics message passing framework that is designed to simplify programming for various robotic applications [Quigley et al. 2009]. By incorporating ROS support within the simulation framework, we natively support software packages written for sensors and actuators, and we can avoid a separate API development. Thus, OpenUAV enables its users to take advantage of the ROS community developed packages and reduces the code changes necessary when transitioning from the simulation environment to real-world deployment.
Figure 19. Components of an OpenUAV container. The simulation components include Gazebo, Unity3D, ROS, PX4 and QGroundControl. The user interactive components consists of NoVNC, TurboVNC and SSH/SSHFS. The NoVNC and SSHFS are accessible on the local machine using the container’s IP address with orts 40001, 22 respectively.

3.2.1.3 PX4

PX4 is an open-source flight control software for UAVs and other unmanned vehicles [Meier, Honegger, and Pollefeys 2015]. It provides a large variety of aerial, ground and sea vehicles, as well as basic and advanced flight control capabilities like position, velocity and rate controllers, state estimators, various sensor support, flight logging, and remote flight control systems. Additionally, the latest version of PX4 provides higher level autonomous capabilities like collision avoidance and terrain relative navigation. Communication and control between the user’s ROS code and the vehicle are simplified by MAVROS, which is a ROS package that converts the communication to MavLINK protocol. Hence, through ROS, we can obtain necessary sensor topics, state estimator values, and publish vehicle actuator commands.
3.2.1.4 QGroundControl

QGroundControl is a remote flight monitoring and planning software that communicates to any MavLINK enabled vehicles [Zurich 2013]. Our simulation environment includes QGroundControl to reflect actual field experiments, where a ground control software is essential for PX4 enabled vehicles. It provides data-logging capabilities that are invaluable for analyzing and evaluating the simulations.

3.2.1.5 Unity3D Game Engine

Unity is a cross-platform game engine primarily used for the development of 2D, 3D, and virtual reality games [“Unity Technologies” 2020]. Unity3D provides photo-realistic rendering due to its art assets, with material properties for shadows, specularity, shading, and textures for the game objects in the scene. Unity also has algorithms for occlusion culling, which disables the rendering of objects that are not currently seen by the camera. Unity3D photorealism provides the necessary outdoor environment to test the autonomy code written in the ROS-Gazebo framework. Hence the perception of the cameras is rendered through the Unity3D scene while still maintaining the physics from Gazebo. We achieve this by transmitting the pose and twist messages of all Gazebo robot models to its corresponding pose in Unity3D and disabling Unity3D’s physics and collisions for those models. Since Unity3D does not have a metric unit, we need to measure each game object’s dimensions from the mesh file. Furthermore, we must follow the metric units in Unity3D for all objects in the scene to avoid any discrepancies with pose and twist messages from Gazebo.

The OpenUAV utilizes ROS-Sharp to communicate between ROS master and
In OpenUAV, we utilize operating-system-level virtualization to deliver software in a separate file, network, and process namespace usually referred to as containers.
Figure 21. Unity3D rendering of a volcanic plume in OpenUAV. This simulation environment is used to study mapping and sampling strategies for volcanology research.

3.2.2.1 Containerization (Docker)

Container technology has become the preferred means of packaging and deploying applications. Docker orchestration tool provides a mechanism to package the software, its libraries, and its dependencies into a single lightweight container. Out of the box, containerization provides isolation from the host as well as from other containers. It also improves the security of the application by restricting the possible host system calls, providing isolated namespaces, and running applications in their least privileged mode. All OpenUAV containers have a shared network namespace, and a Xorg server with GPU to do 3D renderings. Another advantage of containers is its ability to configure resources allocated to each container. These restrictions are useful for
replicating actual vehicle compute power in the simulation containers by having similar memory constraints and compute capacity.

3.2.2.2 NGINX

Nginx is a high-performance web server that implements load balancing, reverse proxying at the server-side, and HTTP caching [Reese 2008]. OpenUAV utilizes Nginx as a reverse proxy to access different remote desktop sessions.

A user is provided with a unique simulation ID when they create an account in OpenUAV. The unique ID is used to form a remote desktop URL such as term-<simulation ID>.openuav.us, which researchers and students use to access their simulations. Nginx also acts as a TCP streaming proxy for the OpenSSH service inside the container. OpenSSH service and SSHFS (SSH FileSystem) enable the sharing of container filesystem to the user’s local machine. This feature enables users to develop software remotely and execute code. However, we enable streaming ports as per user requirements on the OpenUAV server, since it takes up a port on the server. Figure 22 shows the overall system architecture of the OpenUAV.

3.2.3 Interactive components

In the following section, we describe the software components that allow users to interact with the simulations running inside the containers. Remote computing and High-Performance Computing (HPC) utilize these services.
3.2.3.1 TurboVNC

When used with VirtualGL, TurboVNC provides a high performing and robust solution for displaying 3D applications over different types of networks [Deboosere et al. 2007; Cambridge 2019]. TurboVNC’s JPEG compression algorithm provides fast rendering of the Lubuntu desktop session and the applications to any of the connected viewers. VirtualGL is used to send the 3D OpenGL rendering commands directly to the X Server with GPU (not through TurbVNC), and the generated 2D framebuffer image in GPU memory is displayed on TurboVNC using the XPutImage command.

3.2.3.2 NoVNC

NoVNC is a JavaScript-based Virtual Network Computing (VNC) application that provides VNC sessions over the web browser [Martin et al. 2015]. NoVNC follows the standard VNC protocol and has support for persistent connection through WebSockets. The containers have NoVNC client to connect with the TurboVNC server. The NoVNC displays the VNC session over port 40001. Nginx proxies this URL to the correct simulation container. With NoVNC, users can switch between multiple machines by quickly accessing the simulation URL in their browsers. A user can connect a VNC client directly to the TurboVNC’s VNC session. However, this requires exposing the VNC session over Nginx as a TCP streaming proxy.
3.2.3.3 Secure Shell FileSystem (SSHFS)

Secure Shell FileSystem allows users to mount a remote file system using the Secure Shell File Transfer Protocol (SFTP) [Hoskins 2006]. Through SSHFS, the OpenUAV framework provides a secure remote file access mechanism to simulation directories by having it mounted as a remote filesystem in the user’s local machine. SSHFS equips researchers to develop the code on this synchronized directory, thereby utilizing Integrated Development Environments (IDE) on the local machine.

3.2.4 Academic Applications

The OpenUAV framework was used by 30 students as part of a robotics course at Arizona State University (ASU) and for Virtual 2020 NSF Student CPS Challenge
Figure 23. OpenUAV used as part of the SES 494/598 Autonomous Exploration Systems course. On the left, instructors can view every student’s simulations through the CPS-VO dashboard. Login names have been partially masked to maintain user anonymity. On the right, it shows a student engagement, where the student explores different simultaneous localization and mapping packages.

during COVID-19 pandemic [CPSChallenge 2020]. Cyber-Physical Systems Virtual Organization (CPS-VO) is a collaboration among CPS professionals in academia, government, and industry [CPS-VO 2020]. Each student creates an account on the CPS-VO page and uses it as the authentication to create simulation containers in the OpenUAV framework. To maintain a judicious usage of CPU and GPU resources on the machine, the CPS-VO lets students suspend a simulation and resume back from the saved simulation. “Suspend” and “Resume” functions in CPS-VO are implemented using the Docker’s pause and unpause feature. A single OpenUAV server can natively handle multiple CPS-VO like integrations, through different Nginx server blocks. Moreover, the OpenUAV testbed saves the student’s container images every night as a backup to avoid loss of data. Figure 23 demonstrates a student’s engagement of OpenUAV testbed as part of the coursework.
4.1 Conclusion

We demonstrated a visually realistic underwater simulation environment utilizing Gazebo physics and Unity3D camera rendering. We approached the problem of coordinated navigation and localization of an AUV with an ASV in this new simulation environment. By having a fiducial marker underneath the vehicle, we were able to calculate the pose of the AUV with respect to the ASV. These pose observations were filtered using a Kalman filter, and the AUV’s controller used the estimated poses for achieving coordinated navigation.

We demonstrated a simulation testbed useful for field robotics, that natively supports PX4 and QGroundControl. We described how photorealism is achieved in this testbed using the Unity3D game engine and ROS-Sharp. Moreover, the testbed was deployed as part of a robotics course work to understand student and academic requirements.

4.2 Simulation Limitations

There are limitations to our approach of using Gazebo physics and Unity3D perception through ROS-Sharp. The significant limitations encountered are as follows:

- The communication latency added by ROS-Sharp for Unity3D perception is
<table>
<thead>
<tr>
<th>Simulation Environment</th>
<th>Number of cameras</th>
<th>Average Images/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS-Sharp</td>
<td>1</td>
<td>29.153</td>
</tr>
<tr>
<td>ROS-Sharp</td>
<td>2</td>
<td>15.340</td>
</tr>
<tr>
<td>ROS-Sharp</td>
<td>3</td>
<td>11.806</td>
</tr>
<tr>
<td>ROS-Sharp</td>
<td>4</td>
<td>9.262</td>
</tr>
<tr>
<td>ROS-Sharp</td>
<td>5</td>
<td>7.057</td>
</tr>
<tr>
<td>ROS-Sharp</td>
<td>6</td>
<td>5.745</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>1</td>
<td>27.220</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>2</td>
<td>18.219</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>3</td>
<td>11.596</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>4</td>
<td>7.446</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>5</td>
<td>6.198</td>
</tr>
<tr>
<td>ROS-Sharp and Gazebo simulation</td>
<td>6</td>
<td>5.211</td>
</tr>
</tbody>
</table>

Table 3. Evaluation of Publish Rate for 640x480 Images

...a concern when developing applications involving both Unity3D and Gazebo environments. In Table 3, we evaluated the performance of basic ROS-Sharp communication with a ROS-Sharp and Gazebo underwater physics world. The evaluation results show a gradual decrease in image throughput as we increase the number of cameras. This reduction in image throughput with an increasing number of cameras is an issue because this will not allow us to do swarm simulations. Moreover, the overhead of running Gazebo and ROS-Sharp resulted in decreased throughput compared with a basic ROS-Sharp setup.

- The Vulkan API is a cross-platform 3D graphics library used by Unity3D to compute low-overhead 3D graphics. Since Nvidia containers do not support Vulkan API, our OpenUAV containers do not have the Vulkan API support. The consequences of missing Vulkan API include frequent failures during simulation of high amplitude Gerstner wave simulations in Unity3D.

- We found GUI bugs when running Unity3D in containers. For instance, navigating the primary camera over the scene frequently causes crashes [Anand 2020]. This has a negative impact on developer’s productivity.
• It is challenging to model interactive environments using our Gazebo physics/Unity3D perception approach. As shown in the AUV-ASV scenario, the ASV images do not contain the Kelvin wake pattern or water sloshing.
• This simulation environment does not currently support depth cameras, which are extensively used in robotics applications.

4.3 Future Work

Extensive pool and field experiments have to be conducted in Hawaii to evaluate the ASV-AUV system’s performance. To achieve robust autonomy for underwater vehicles, we need to develop controller methods and approaches for achieving terrain aware navigation and collision avoidance. This simulation architecture can be used for understanding underwater photogrammetry challenges, developing various methods to remove water from underwater images without deploying the vehicle in water [Akkaynak and Treibitz 2019].

Furthermore, to save our coral reefs, we need to achieve swarm autonomy for underwater vehicles. Soon, we envision a future where a swarm of underwater vehicles mapping our coral reefs and leading the conservation efforts, as shown in Figure 24.

We presented OpenUAV as a scalable, extensible, and user-friendly simulation framework that saves users from the time-consuming software configurations and setups. We hope to utilize this framework to support research in planetary and space science and fluid dynamics in the future. Through Unity3D support, we can create a perception for extreme environments such as marine ecosystems, volcanic environments, and render planetary rover environments such as lunar and martial
Figure 24. A future where a swarm of AUVs collects coral reef data and geo-reference the data using an ASV. The Global Airborne Observatory calibrates its sensors based on the ASV-AUV field data, to achieve a large scale mapping of reefs.
environments. In the future, we plan to achieve support for the following features in OpenUAV:

- Support depth cameras in Unity3D and Gazebo simulations.
- Use hardware-in-the-loop simulations with real cameras rendering the Unity3D world.
- Develop OpenUAV simulations through API calls that can be used by robotics packages as continuous integration.
- Further improve OpenUAV to support the requirements of academic students and research community.
REFERENCES


