HoloOcean: An Underwater Robotics Simulator

Easton Potokar, Spencer Ashford, Michael Kaess, and Joshua G. Mangelson

***Abstract*— Due to the difficulty and expense of underwater field trials, a high fidelity underwater simulator is a necessity for testing and developing algorithms. To fill this need, we present HoloOcean, an open source underwater simulator, built upon Unreal Engine 4 (UE4). HoloOcean comes equipped with multi-agent support, various sensor implementations of common underwater sensors, and simulated communications support. We also implement a novel sonar sensor model that leverages an octree representation of the environment for efficient and realistic sonar imagery generation. Due to being built upon UE4, new environments are straightforward to add, enabling easy extensions to be built. Finally, HoloOcean is controlled via a simple python interface, allowing simple installation via pip, and requiring few lines of code to execute simulations.**

I. Introduction

An effective simulation environment for autonomous underwater vehicles (AUVs) can accelerate the development of algorithms and applications. This is true for all robotics systems, but is particularly necessary for AUVs where field testing is expensive and high-risk.

There are many modern day applications of AUVs that can dramatically improve scientific knowledge, quality of life, and safety. These applications include inspection of marine infrastructure such as dams, ship hulls, and communication lines, as well as exploration of oceans that can lead to discoveries in the fields of geology, marine biology, and medicine. However, all these applications require complex algorithms to be designed and tested, which can be costly and unreasonably challenging without a virtual environment on which to first develop them.

We present HoloOcean, an open source underwater simulator. It is built upon the reinforcement learning simulator Holodeck [1] and the capabilities of Unreal Engine 4 (UE4) [2]. UE4 in particular provides accurate simulation dynamics, high-fidelity imagery, a mature environment construction editor, and a C++ interface to add custom sensors, agents, etc. Specifically, HoloOcean has the following features:

1) A simple python interface, allowing for quick installation and effortless use.
2) Ease of adding new environments. UE4 is a well documented game building engine with many marine and underwater assets already made.
3) A novel and efficient imaging sonar implementation built upon an octree structure that results in realistic sonar imagery.
4) Full support for multi-agent missions, including implemented acoustic and optical modem sensors for realistic cooperative simulations.

This work relates to Department of Navy awards N00014-21-1-2435 and N00014-21-1-2272 issued by the Office of Naval Research.

E. Potokar, S. Ashford, and J. Mangelson, are at Brigham Young University; M. Kaess is at Carnegie Mellon University.
{eastonpots, spencerashford, joshua_mangelson}@byu.edu, kaess@cmu.edu

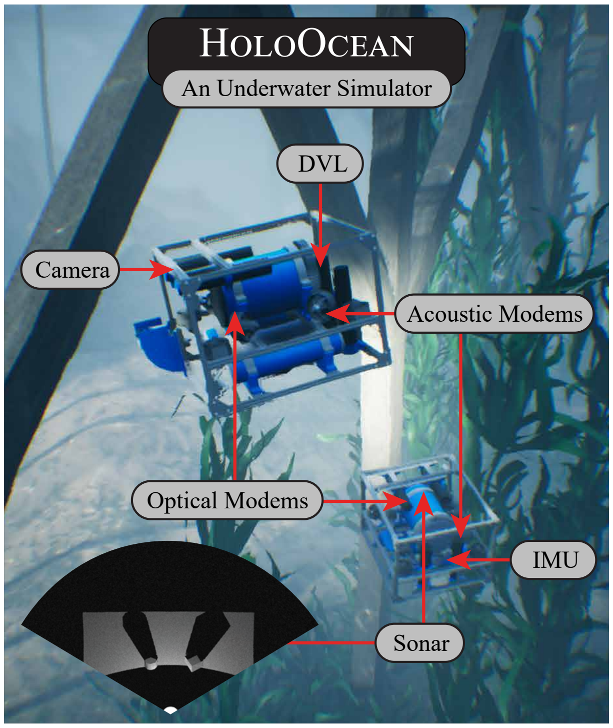


Fig. 1: HoloOcean is an open source underwater simulator, built upon Unreal Engine 4. It comes with a variety of common underwater sensors such as a DVL, IMU, optical and acoustic modems, and an imaging sonar.

The paper is organized as follows. Section II reviews current underwater robotics simulators and sonar sensor models. In Section III, we describe our novel sonar imagery algorithm, followed by Section IV where we review the other various implementations and measurement models for common underwater sensors such as a Doppler velocity log (DVL), inertial measurement unit (IMU), GPS, camera, and depth sensor. Multi-agent missions and communications via optical and acoustic modems are described in Section V. Customization of missions and the simple python interface is laid out in Section VI. Various environments made in UE4 are shown in Section VII, and Section VIII summarizes the article and proposes future work.

II. Related Work

Creating a realistic underwater simulator requires many features to be useful for algorithm development including, but not limited to, multi-agent support; realistic sensor simulations; accurate underwater dynamics; ease of use; integration with existing systems; and, preferably, open source [3]. Another requirement is a lack of heavy dependencies. Heavy

Simulator	Multi-Agent	Communications	Sonar	Open Source	Small Dependencies	Maintained	Simple Mission Setup
HoloOcean	✓	✓	✓	✓	✓	✓	✓
UUV Simulator	✓	×	∗	✓	×	∗	✓
UWSim	✓	×	∗	✓	×	×	×
MarineSim	✓	×	×	×	×	×	∗

TABLE I: Comparison of common underwater simulators. ✓denotes that the simulator has a feature, $*$ that it has a limited implementation or is unknown, and $\times$ that it does not. There are many other common robotics simulators that are not listed here, but most don't have support for underwater robotics.

Algorithm 1: Sonar Imagery Computation

```
1  Load ≜ Make & cache if not made, else load cache;
2  Unload ≜ Delete from memory;
3  FOV ≜ Field of view;
4  R ≜ RootNode;
5  M represents a mid-level octree node;
6  L represents a leaf;
7  Load R till Ms;
8  Load all M within range;
9  foreach Sonar Tick do
10     foreach M_i not in (R and FOV) do
11         Unload M_i;
12     foreach M_i in (R and FOV) do
13         Load M_i;
14         foreach L_j in (M_i and FOV) do
15             $d_{L_j} = n_{L_j} \cdot n_{imp}$;
16             if $d_{L_j} > 0$ then
17                 Add L_j to corresponding $\phi, \theta$ bin;
18     foreach $\phi, \theta$ bin do
19         Sort bin from closest range to farthest;
20         Take first cluster of elements in bin;
21         foreach L_j in cluster do
22             Add L_j to corresponding $\phi, r$ bin;
23     foreach $\phi, r$ bin do
24         $z_{ij} = \left(\frac{1}{n}\sum_{k=0}^{n} d_k\right)(1 + \mathrm{w}^{sm}) + \mathrm{w}^{sa}$;
25     return z;
```

Fig. 2: Pseudocode of our sonar imagery algorithm. Lines 10-17 corresponding to recursively searching our octree to find leaves in our field of view, lines 18-22 correspond to removing leaves that may be in shadows, and lines 23-24 to the final computation of the image.

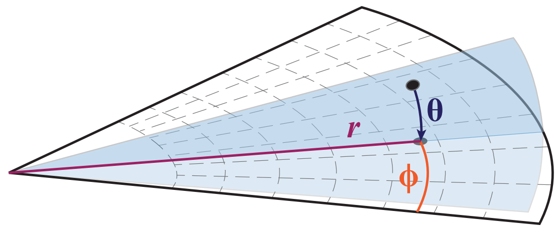


Fig. 3: Geometry of an imaging sonar. Shown are a single beam and the elevation θ, azimuth ϕ, and range r of a single point. All elevation data will be lost as objects are projected onto the azimuth-range plane and then binned accordingly.

dependencies, such as Robot Operating System (ROS) [4], can make installation and use cumbersome, particularly when the simulator is only being used for data generation. Various attempts at these features are listed in Table I, but as far as we know there are none that match all of these criteria.

UUV Simulator [5] is one of the more mature options, built upon the popular open source robotics simulator Gazebo [6]. It has accurate modeling of hydrostatic and hydrodynamic effects, multi-agent support, a preliminary sonar implementation [7], and is easily configurable. However, it requires the installation of ROS, lacks multi-agent communications, and does not appear to be actively maintained.

UWSim [8] is built on OpenSceneGraph and osgOcean [9]. It also has multi-agent support and is open source, but depends on ROS, is difficult to configure, is not being actively maintained, and has a sonar model that is more akin to a LiDAR.

Built upon USARSim [10], MarineSIM [11] is another simulator for underwater navigation, but is not open source, which limits the possibility for future development. Other work in USARSim includes an implementation of acoustic multi-agent communications [12], but lacks other common underwater sensors.

An efficient and accurate implementation of a multi-beam imaging sonar is also essential for research in underwater perception and localization. The UUV Simulator sonar model leverages a simulated depth camera and GPU computations [7, 13] that appears promising, but has drawbacks due to the depth camera field of view not matching that of a true imaging sonar. Others leverage ray tracing [14] or more accurate acoustic physics [15], which can be computationally inefficient and currently have no integration with existing simulators.

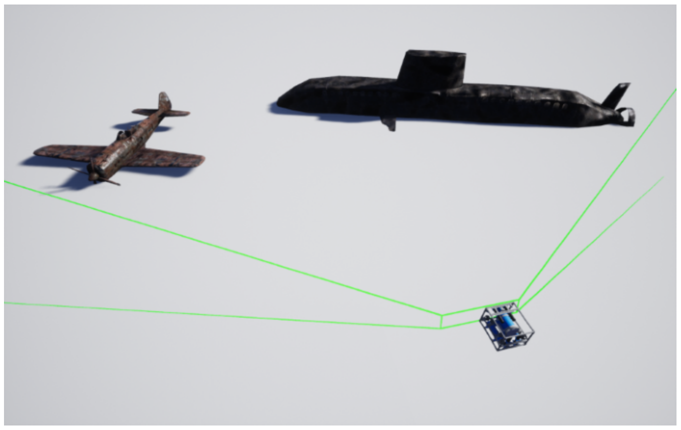
Holodeck [1] is an open source reinforcement learning and robotics simulator. It's built upon UE4 [2], providing it with high-fidelity imagery, accurate dynamics built upon the PhysX physics engine [16], and a mature community with many environment assets already made. Further, Holodeck has a simple python interface, allowing for easy installation and use on a variety of systems. In this work, we propose HoloOcean that builds on Holodeck and augments it with accurate underwater dynamics, multi-agent communications, a realistic imaging sonar implementation, and other underwater sensor models.

III. Imaging Sonar

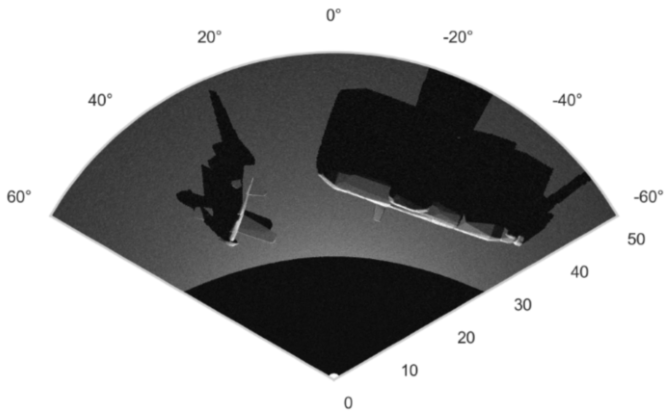
An imaging sonar is a common underwater sensor used to generate imagery of the environment. This imagery can be used for localization, mapping, visualization, or various other algorithms. In this section we cover how an imaging sonar functions and present our algorithm for generating realistic imagery, which is summarized in Fig. 2.

A. *Operation*

Multi-beam imaging sonar sensors use acoustic waves to capture imagery of their environment. A wave is emitted and upon encountering an object, part of the wave is reflected back to the sonar, where intensity is recorded and beamforming techniques are used to determine the direction of the return. This intensity will be dependent on the surface normal vector it encountered as well as the normal of the impacting beam. This is because a surface that is aligned with the beam will reflect more energy than one that is perpendicular. Time



(a)



(b)

Fig. 4: Example of imaging sonar simulation. (a) Shows the environment, with green rays showing an approximation of the field of view of the sonar. The visible range is given by an azimuth of 120°, elevation of 20°, and a range from 1 to 50 meters. An octree resolution of 2cm is used. (b) Shows the resulting sonar image, including minor multiplicative and additive noise. The image has 512 azimuth beams and 1024 range bins.

from sending to receiving the wave is also measured, and from this the range r is calculated using the speed of sound underwater.

The sonar reading in a given horizontal direction forms a beam. Each beam has a horizontal angle, known as the azimuth ϕ, and a vertical width, known as the elevation θ, as seen in Fig. 3.

Each beam will correspond to a column in the resulting 2D image, while each row will correspond to a range interval with its resulting quantity being given by its echo intensity. In this way, a sonar gives a projection of 3D space onto a 2D image, with the elevation as the dimension that is lost.

B. *Projection Model*

To simulate sonar imagery, we discretize the environment using an octree implementation. In each leaf of the octree, both location and surface normals are stored. This octree allows us to project the points onto the sonar image, rather than using expensive ray tracing or relying on the square field of view of a depth camera. When octrees are generated, nodes are only added if the box is only partially occupied. This eliminates the need to store areas of free space, as well as areas inside of objects.

To find the octree leaves in our field of view, we first recursively search the root octree to find all mid-level nodes in our field of view. Note, to allow for imaging of other agents during operation, a small octree is made for each agent and updated after each time step. In parallel, we then recursively search the agent octrees along with all previously found mid-level nodes to find all the leaves in the field of view.

Once the leaves are found, the dot product d between the surface unit normal n_s and impact unit normal n_i of each is calculated. This gives us the cosine of the angle ψ between the two normals as follows,

$$d \triangleq n_s \cdot n_i = \cos(\psi) > 0 \implies |\psi| < \frac{\pi}{2}. \tag{1}$$

If the angle is greater than $\frac{\pi}{2}$, this implies the leaf must be on the backside of an object and it isn't kept. Thus, we keep all leaves with a dot product greater than 0.

C. *Shadows*

Once all leaves on the front side of objects are found, we must remove the ones that lie in shadows. To do this, we first place each of the leaves in their corresponding ϕ, θ bin. We have found that an elevation bin size of $0.03°$ provides the best results.

In parallel, each of these bins is sorted by ascending range. We keep the first cluster of a bin. We do this by iterating through the sorted bin, checking if the difference between two adjacent elements' range value is less than some predefined ϵ. Once a gap larger than ϵ is found, everything after it is then removed, as it is part of a shadow.

There exist more accurate shadowing algorithms, but at a significant computational cost. This algorithm has provided us with a good combination of both accuracy and efficiency.

The resulting leaves are then sorted into their corresponding ϕ, r bins. The following calculation is then performed to determine the pixel value z^s_{ij} of the i, jth bin,

$$\begin{gathered} z^s_{ij} = \Big(\frac{1}{n}\sum_{k=0}^{n} d_k\Big)(1 + \mathrm{w}^{sm}) + \mathrm{w}^{sa} \\ \mathrm{w}^{sm} \sim \mathcal{N}(0, \sigma^{sm}), \quad \mathrm{w}^{sa} \sim \mathcal{R}(\sigma^{sa}), \end{gathered} \tag{2}$$

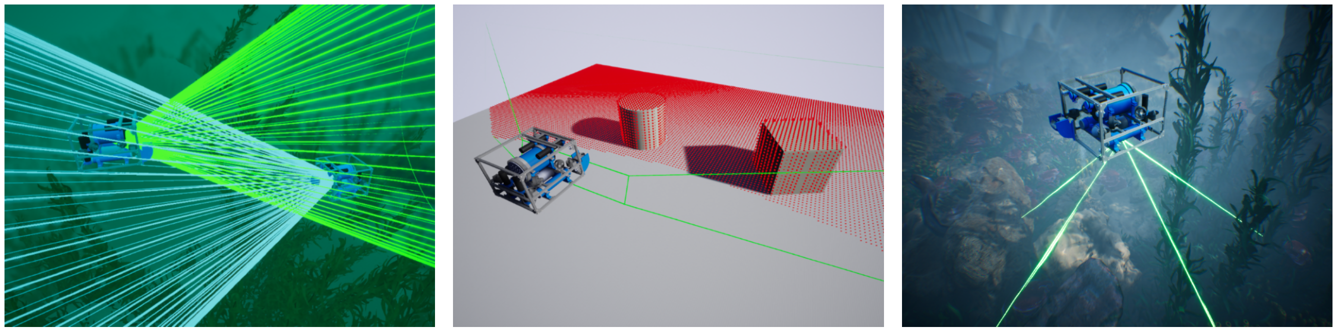
where d_k was computed previously in eqn. (1), $\mathrm{w}^{sa}, \mathrm{w}^{sm}$ provide additive and multiplicative noise, respectively, and $\mathcal{R}$ is the Rayleigh distribution. These steps are all summarized in Fig. 2.

D. *Efficiency & Memory*

Unfortunately, as many of our environments are 2km$\times$2km, generating a full octree representation all at once requires an unwieldy amount of time, and storing it all in memory is impossible. To combat this, we generate, cache, and load octree leaves in real time, as follows.

On startup, the root octree till mid-level nodes is either made and cached, or loaded from file. Then, each mid-level node within some r_{max} of our vehicle that isn't already cached is created and saved to file.

During the recursive search, when a mid-level node is found, we check if it's been loaded; if not, then it's loaded



(a) (b) (c)

Fig. 5: Demonstration of various sensor visualization tools included in HoloOcean. (a) Shows two vehicles transmitting over optical modem, with their available line of sight highlighted as cones. (b) Shows a sonar simulation with an octree generated at 10cm. Highlighted are the field of view of the sonar in green, and each octree leaf inside the field of view in red. (c) Shows the beams of a DVL highlighted in green.

from the cache or made and cached. Each mid-level node that isn't in the field of view is deleted from memory.

We've found that loading from our cache, a directory of json files, is fast and has negligible impact on performance since the number of new mid-level nodes in the field of view each iteration is generally small. In addition, the cache is persistent between simulations, thus it can be reused with each new mission. This method also removes the need to generate the full 2km×2km octree representation on first startup, instead only generating the leaves that are likely to be used.

This process produces realistic sonar images, as can be seen in Fig. 4. At a 5.12m mid-level node size, 2cm minimum leaf size, and 50m max distance we can sample 2 sonar images per second, and at 2.56m mid-level, 1cm minimum, 10m distance we can sample about 14 images per second.

IV. Sensor Models

HoloOcean comes built in with a variety of sensors used in underwater robotics. They've been implemented with real world sensors in mind, and thus come with many configuration options and noise models. Note, all sensors have been configured with their own sensor frame, defined with respect to the robot frame as $R_s^r, \boldsymbol{t}_s^r$. These are represented in the world frame $R_s^w, \boldsymbol{t}_s^w$ as follows,

$$\begin{aligned} R_s^w &= R_r^w R_s^r \\ \boldsymbol{t}_s^w &= R_r^w \boldsymbol{t}_s^r + \boldsymbol{t}_r^w . \end{aligned} \tag{3}$$

Each sensor has various configurations that can be set in a simple json file, as shown in Fig. 6. These configurations include $R_s^r, \boldsymbol{t}_s^r$, the sample rate in Hz, covariance for noise models, visualization tools as seen in Fig. 5, and various other sensor specific configurations. Further, when desired, a Lightweight Communications and Marshalling (LCM) [17] wrapper has been included to allow for an interface with real code. A ROS wrapper could also easily be added and is under active development.

In this section, we cover the measurement model used for each sensor, as well as various other implementation details.

A. Doppler Velocity Log

A DVL functions by sending out four acoustic waves and upon their return, uses incoming and outgoing wave velocities to calculate the sensor velocity in the sensor frame by leveraging the Doppler effect. We denote the angle of the acoustic waves from the negative z-axis as α, and the calculated velocity of the beams as the 4-vector v_b. The sensor measurement z^v is then modeled as,

$$\begin{aligned} z^v &= R_b^s(v_b + \mathrm{w}^v) = R_w^s v_w + R_b^s \mathrm{w}^v, \quad \mathrm{w}^v \sim \mathcal{N}(0, \Sigma^v) \\ R_b^s &= \begin{bmatrix} \frac{1}{2\sin(\alpha)} & 0 & \frac{-1}{2\sin(\alpha)} & 0 \\ 0 & \frac{1}{2\sin(\alpha)} & 0 & \frac{-1}{2\sin(\alpha)} \\ \frac{1}{4\cos(\alpha)} & \frac{1}{4\cos(\alpha)} & \frac{1}{4\cos(\alpha)} & \frac{1}{4\cos(\alpha)} \end{bmatrix} \end{aligned} \tag{4}$$

where w^v is Gaussian noise, R_b^s is the transformation from the beam frame to the sensor frame [18], and v_w is the velocity in the world frame that we pull from UE4.

Additionally, a DVL also calculates the distance each wave traveled, resulting in a 4-vector of ranges that can be used to assemble a sparse point cloud.

B. Inertial Measurement Unit

An IMU sensor measures angular velocity and linear acceleration in the sensor frame. A time-varying bias is commonly found in both measurements of an IMU, and we include it in the sensor model as follows [19],

$$\begin{aligned} z^a &= R_w^s a_w + b^a + \mathrm{w}^a, && \mathrm{w}^a \sim \mathcal{N}(0, \Sigma^a) \\ z^\omega &= \omega + b^\omega + \mathrm{w}^\omega, && \mathrm{w}^\omega \sim \mathcal{N}(0, \Sigma^\omega) \\ b^a &= b^a + \mathrm{w}^{ba}, && \mathrm{w}^{ba} \sim \mathcal{N}(0, \Sigma^{ba}) \\ b^\omega &= b^\omega + \mathrm{w}^{b\omega}, && \mathrm{w}^{b\omega} \sim \mathcal{N}(0, \Sigma^{b\omega}). \end{aligned} \tag{5}$$

HoloOcean can also be configured to return the bias for ground truth purposes in cases where it's also being tracked.

C. Depth Sensor

Underwater pressure is directly proportional to depth, and is often used as a z-axis position measurement in the global frame. We measure it as follows, given our global z position is denoted by p_z [20],

$$z^d = p_z + \mathrm{w}^d, \qquad \mathrm{w}^d \sim \mathcal{N}(0, \sigma^d). \tag{6}$$

D. Camera

An underwater camera is also included that allows for imagery of the environment to be taken from the camera frame. UE4 has long been known for its high-fidelity imagery, providing games with realistic graphics for many years. This

```
{
  "name": "MyMission",
  "world": "PierHarbor",
  "ticks_per_sec": 300,
  "agents": [
    {
      "agent_name": "auv0",
      "agent_type": "HoveringAUV",
      "sensors": [
        {
          "sensor_type": "IMUSensor",
          "socket": "IMUSocket",
          "location": [1.0, 2.0, 3.0],
          "rotation": [1.0, 2.0, 3.0],
          "Hz": 300,
          "configuration": {
            "Sensor specific config"
          }
        }
      ],
      "location": [1.0, 2.0, 3.0],
      "rotation": [1.0, 2.0, 3.0],
    }
  ],
  "window_width":  1280,
  "window_height": 720
}
```

Fig. 6: Example of a mission configuration. The sensor socket is a predefined frame on the vehicle body where a sensor can be placed. The sensor location and rotation parameters are then used as an offset from the chosen socket.

can be fine tuned to give photorealistic underwater imagery [21], as is the case in HoloOcean.

E. GPS Sensor

While GPS is not available underwater, when near the surface AUVs can gain connection and receive GPS measurements to aid in localization. We model this by the following, letting global position be denoted as p,

$$z^g = p + \mathrm{w}^g, \qquad \mathrm{w}^g \sim \mathcal{N}(0, \Sigma^g). \tag{7}$$

Further, we only allow GPS measurements to be received when within a certain distance d of the surface, where d can be configured to be a random variable distributed as $\mathcal{N}(d, \Sigma)$.

F. Pose Sensor

A pose sensor is also included for use as ground truth. It returns an element of $SE(3)$, of the form,

$$z^p = \begin{bmatrix} R_s^w & \boldsymbol{t}_s^w \\ 0_{1\times 3} & 1 \end{bmatrix}. \tag{8}$$

V. Multi-Agent Missions & Communications

HoloOcean also allows for an arbitrary number of agents to be used in a scenario. Adding agents, with any kind of sensors attached, is as simple as adding a few lines to a json file. HoloOcean comes with two agents models: our in-house custom hovering AUV based on the BlueROV2 by BlueRobotics, and a generic AUV. Both follow a simple buoyancy dynamics model, while the hovering AUV has forces applied to the thruster locations, and the generic AUV has a thruster along with fin dynamics as defined in [22].

In many multi-agent missions, communications between agents is essential for cooperative efforts. For underwater robotics, this becomes a difficult task as communications are restricted to optical or acoustic modems. To better simulate these scenarios, these modems are also included in HoloOcean as described below.

A. Acoustic Modem

Our acoustic modem model is based off of the capabilities of the Blueprint Subsea SeaTrac X150 [23]. This means that to communicate, an acoustic wave is sent from one beacon to another. In HoloOcean, when a message is sent from an acoustic modem, it's sent with a message type along with its data payload. We check if any messages are sent at the same time; if there are, all messages are dropped to simulate interference of acoustic waves.

We calculate how many time steps it will take for the acoustic message to arrive. Upon arrival, the payload is received, and depending on the message type, a return message is possibly sent. Along with the payload, other data is received such as bearing, range, and depth as specified by the message type.

B. Optical Modem

The optical modem is roughly based on the Hydromea LumaX [24]. It operates similarly to the acoustic modem, with a few notable differences. In HoloOcean, when an optical modem sends a message, it first checks to make sure the sending and receiving beacons are oriented properly to see each other and that no object is obstructing the view.

Once a connection is verified, the message and its payload are sent, in this case with no delay.

VI. Python Interface

HoloOcean is essentially built as a python wrapper of an UE4 compiled game binary. This allows for a simple python interface for controlling simulations, as well as easy installation.

A. Installation

Installation is performed by installing the python package, then from the python package a simple command will download the UE4 packaged binary. Note, as more environments for HoloOcean become available, each environment will be available for download individually as an UE4 packaged binary. Installation is as follows. First from the command line to install the python package while using python 3.6+,

```
$ pip install holoocean
```

Then from a python console or script to download and install the UE4 environments,

```
import holoocean
holoocean.install("Ocean")
```

B. Python Interface

HoloOcean's python interface is modeled after OpenAI Gym [25]. This means controlling the robots can be done in a few lines of code. For example, the following code creates the predefined mission "PierHarbor-Hovering" which includes our custom in-house Hovering AUV loaded with its sensors at the pier environment. It then sends the AUV commands, one for each of its eight thrusters, to push it

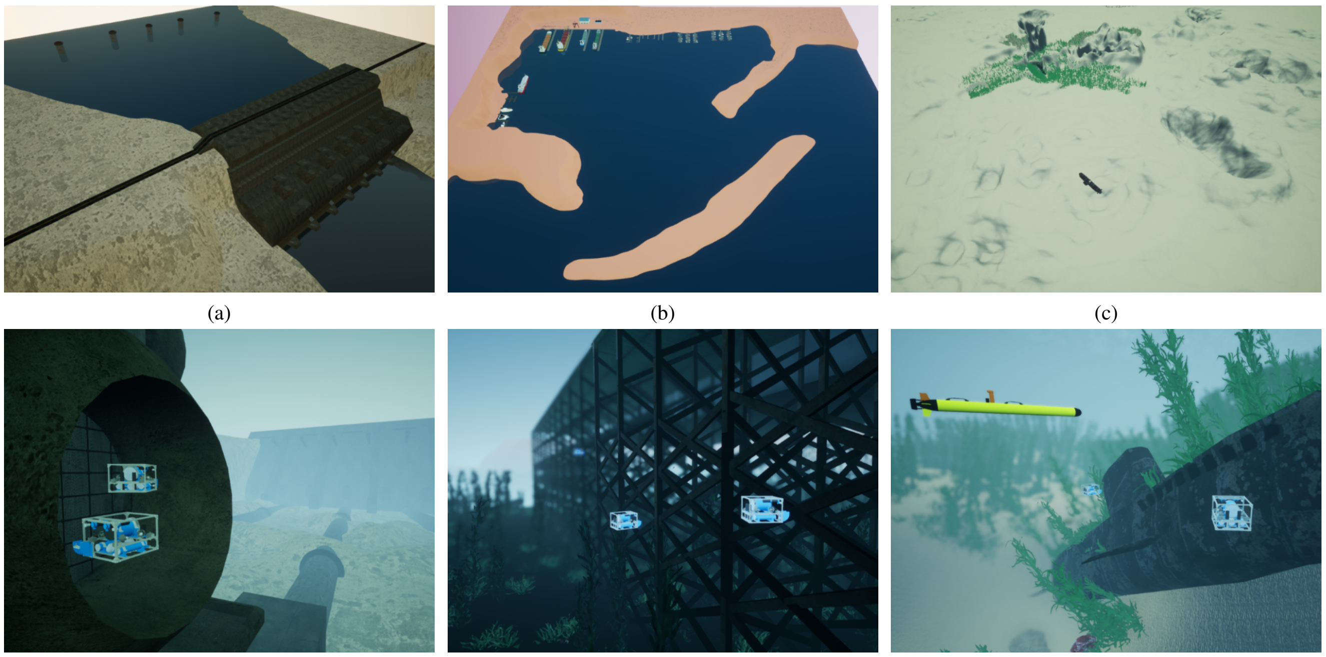


Fig. 7: Examples of the environments that we have created in UE4 for use in HoloOcean. (a) Shows the full dam environment with size 650m$\times$650m, with (d) showing a closeup of one of the pipes. (b) Shows an overview of the pier environment, including all 3 sizes of piers, with a total size of 2km$\times$2km, and (e) showing a closeup of a pier in the top left of (b). Finally, (c) shows the open water environment which includes a number of sunken submarines and planes for inspection with a total size of 2km$\times$2km, with one of the submarines shown in (f).

upwards for 200 ticks. At the end of each tick, a "state" dictionary is returned that has all the sensor information sampled at that time step.

```python
import holoocean

env = holoocean.make("PierHarbor-Hovering")

command = [10,10,10,10,0,0,0,0]
for _ in range(200):
    env.act("auv0", command)
    state = env.tick()
```

C. *Configuring Missions*

Configuring missions is easily done by defining a custom configuration in json format, as shown in Fig. 6. Note, the configuration takes in a possible array of agents, as well as an array of sensors objects for each agent. This json, either loaded from a file or inserted directly in the python code, is passed to HoloOcean for mission creation. This allows for painless customization of missions, with any variation of sensors and agents as required.

HoloOcean also comes with a number of other features such as headless mode, configurable time steps, frame rate capping, and others. For further information, documentation is available at `holoocean.readthedocs.io`.

VII. Environments

To provide a wide range of realistic scenarios, we have built various environments for HoloOcean as well. Currently, these include a dam, pier, and open water environment. We have configured each of these with realistic underwater imagery, as well as large underwater areas for use in multi-agent missions. Examples can be seen in Fig. 7.

As one of the most popular game engines, UE4 has a large marketplace full of environments and meshes for purchase by independent users at reasonable prices. This makes rapid development of high quality environments possible. In addition, UE4 has great documentation and a large community behind it, resulting in an abundance of online resources for new users, significantly lessening the initial learning curve.

HoloOcean has already aided us in our research, providing realistic data for verification of the invariant extended Kalman filter for underwater navigation [26].

VIII. Conclusion

Building upon Holodeck and UE4, we created a new open source underwater simulator, allowing for painless building of custom environments from UE4. It also features a simple to install and use python interface. Further, we have implemented a novel imaging sonar model, allowing for realistic real-time imagery. Several common underwater sensors such as a DVL, IMU, camera, GPS, and depth sensor have also been implemented. Multi-agent missions and communications are also supported through acoustic and optical modems. All together, this results in a mature underwater simulator that's extensible and easy to use. Future work will include additional sonar sensors, a GPU implementation of our imaging sonar algorithm, a ROS wrapper to publish sensor data, additional agents including autonomous surface vehicles, more accurate sensor noise simulations, and additional custom environments.

References

[1] J. Greaves, M. Robinson, N. Walton, M. Mortensen, R. Pottorff, C. Christopherson, D. Hancock, J. Milne, and D. Wingate, "Holodeck: A high fidelity simulator," https://github.com/BYU-PCCL/holodeck, 2018.

[2] Epic Games, "Unreal engine," https://www.unrealengine.com, 2019.

[3] D. Cook, A. Vardy, and R. Lewis, "A survey of AUV and robot simulators for multi-vehicle operations," in *Proc. IEEE/OES Autonomous Underwater Vehicles Conf.*, Oct. 2014, pp. 1–8.

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

[5] 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," in *Proc. IEEE/MTS OCEANS Conf. Exhib.*, Sep. 2016, pp. 1–8.

[6] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots and Syst.*, vol. 3, Sep. 2004, pp. 2149–2154 vol.3.

[7] R. Cerqueira, T. Trocoli, G. Neves, S. Joyeux, J. Albiez, and L. Oliveira, "A novel GPU-based sonar simulator for real-time applications," *Computers & Graphics*, vol. 68, pp. 66–76, Nov. 2017.

[8] M. Prats, J. Pérez, J. J. Fernández, and P. J. Sanz, "An open source tool for simulation and supervision of underwater intervention missions," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots and Syst.*, Oct. 2012, pp. 2577–2582.

[9] "osgOcean," https://github.com/kbale/osgocean, 2018.

[10] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: A robot simulator for research and education," *IEEE Robot. and Automation Letters*, pp. 1400–1405, Apr. 2007.

[11] P. Namal Senarathne, W. S. Wijesoma, K. W. Lee, B. Kalyan, M. Moratuwage, N. M. Patrikalakis, and F. S. Hover, "MarineSIM: Robot simulation for marine environments," in *Proc. IEEE/MTS OCEANS Conf. Exhib.*, 2010, pp. 1–5.

[12] A. Sehgal and D. Cernea, "A multi-AUV missions simulation framework for the USARSim robotics simulator," in *Mediterranean Conference on Control and Automation*, 2010, pp. 1188–1193.

[13] D.-H. Gwon, J. Kim, M. H. Kim, H. G. Park, T. Y. Kim, and A. Kim, "Development of a side scan sonar module for the underwater simulator," in *2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, Jun. 2017, pp. 662–665.

[14] K. J. DeMarco, M. E. West, and A. M. Howard, "A computationally-efficient 2D imaging sonar model for underwater robotics simulations in Gazebo," in *Proc. IEEE/MTS OCEANS Conf. Exhib.*, Oct. 2015, pp. 1–7.

[15] A. Rascon, "Forward-looking sonar simulation model for robotic applications," Thesis, Monterey, CA; Naval Postgraduate School, Sep. 2020.

[16] NVIDIA, "Physx," https://github.com/NVIDIAGameWorks/PhysX.

[17] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight Communications and Marshalling," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots and Syst.*, Oct. 2010, pp. 4057–4062.

[18] J. Y. Taudien, "Doppler velocity log algorithms: detection, estimation, and accuracy," Ph.D. dissertation, Pennsylvania State Universiy, 2018.

[19] R. Hartley, M. Ghaffari, R. M. Eustice, and J. W. Grizzle, "Contact-aided invariant extended Kalman filtering for robot state estimation," *Int. J. Robot. Res.*, vol. 39, no. 4, pp. 402–430, 2020.

[20] S. Aravamudhan and S. Bhansali, "Reinforced piezoresistive pressure sensor for ocean depth measurements," *Sensors and Actuators A: Physical*, vol. 142, no. 1, pp. 111–117, 2008.

[21] T. Manderson, I. Karp, and G. Dudek, "Aqua underwater simulator," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots and Syst.*, 2018.

[22] Z. J. Harris and L. L. Whitcomb, "Preliminary Evaluation of Cooperative Navigation of Underwater Vehicles without a DVL Utilizing a Dynamic Process Model," in *Proc. IEEE Int. Conf. Robot. and Automation*, 2018, pp. 4897–4904.

[23] Blueprint, "SeaTrac X150 USBL Transponder Beacon," https://www.blueprintsubsea.com/pages/product.php?PN=BP00795, 2020.

[24] Hydromea, "LumaX," https://www.hydromea.com/underwater-wireless-communication/, 2020.

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *CoRR*, vol. abs/1606.01540, 2016.

[26] E. Potokar, K. Norman, and J. G. Mangelson, "Invariant Extended Kalman Filtering for Underwater Navigation," *IEEE Robot. and Automation Letters*, vol. 6, no. 3, pp. 5792–5799, Jul. 2021.