HoloOcean Documentation

Release 0.5.0

Joshua Greaves, Max Robinson, Nick Walton

Mar 23, 2022

HOLOOCEAN DOCUMENTATION

HoloOcean is a realistic underwater robotics simulator with multi-agent missions, various underwater sensors including a novel imaging sonar sensor implementation, easy installation, and simple use. It's a fork of Holodeck, a high-fidelity reinforcement learning simulator built on Unreal Engine 4.

CHAPTER

ONE

INSTALLATION

HoloOcean is installed in two portions: a client python library (`holoocean`) is installed first, which then downloads world packages. The python portion is very small, while the world packages ("binaries") can be several gigabytes.

1.1 Requirements

- >= Python 3.6
- Several gigabytes of storage
- pip3
- Linux or Windows 64bit
- Preferably a competent GPU
- For Linux: OpenGL 3+

1.2 Stable Installation

The easiest installation is via the pypi repository, done as,

```
pip install holoocean
```

And then to install the binary, simply run

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

Or as a single console command,

```
python -c `import holoocean; holoocean.install("Ocean")`
```

Note: There is a bug on Windows with the package `pywin32` that occurs occasionally. If you see "ImportError: DLL load failed while importing win32event: The specified module could not be found.", it can be fixed by running `pip install pywin32==225`

1.3 Development Installation

Note: If you had previously installed holoocean before moving to the develop or any other branch, then you will need to uninstall the binary by running `holoocean.remove("Ocean")` before proceeding. Failure to do so may result in unpredictable behavior.

To use the latest version of HoloOcean, you can install and use HoloOcean simply by cloning the bitbucket.org/frostlab/holoocean, and ensuring it is on your `sys.path`.

The `master` branch is kept in sync with the pip repository, the `develop` branch is the bleeding edge of development.

To install the develop branch, simply run

```
git clone https://bitbucket.org/frostlab/holoocean/
cd holoocean
git checkout develop
pip install .
```

Then to install the most recent version of the oceans package, run the python command

```
import holoocean
holoocean.install("Ocean", branch="develop")
```

Or as a single console command,

```
python -c `import holoocean; holoocean.install("Ocean", branch="develop")`
```

Note you can replace "develop" with whichever branch of HoloOcean-Engine you'd like to install.

1.4 Docker Installation

HoloOcean's docker image is only supported on Linux hosts.

You will need `nvidia-docker` installed.

The repository on DockerHub is frostlab/holoocean.

Currently the following tags are availible:

- `base` : base image without any worlds
- `ocean` : comes with the Ocean package preinstalled
- `all/latest` : comes with the all packages pre-installed

This is an example command to start a holodeck container

```
nvidia-docker run --rm -it --name holoocean frostlab/holoocean:ocean
```

Note: When ran in docker, HoloOcean can only be ran headless. This means you must pass the parameter `show_viewport=False` into `holoocean.make()`.

Note: HoloOcean cannot be run with root privileges, so the user `holooceanuser` with no password is provided in the docker image.

1.5 Managing World Packages

The `holodeck` python package includes a *Package Manager* that is used to download and install world packages. Below are some example usages, but see *Package Manager* for complete documentation.

1.5.1 Install a Package Automatically

```
>>> from holoocean import packagemanager
>>> packagemanager.installed_packages()
[]
>>> packagemanager.available_packages()
['Ocean']
>>> packagemanager.install("Ocean")
Installing Ocean ver. 0.1.0 from https://robots.et.byu.edu/holo/Ocean/v0.1.0/Linux.zip
File size: 1.55 GB
|| 100%
Unpacking worlds...
Finished.
>>> packagemanager.installed_packages()
['Ocean']
```

1.5.2 Installation Location

By default, HoloOcean will install packages local to your user profile. See *Package Installation Location* for more information.

1.5.3 Manually Installing a Package

To manually install a package, you will be provided a `.zip` file. Extract it into the `worlds` folder in your HoloOcean installation location (see *Package Installation Location*)

Note: Ensure that the file structure is as follows:

```
+ worlds
+-- YourManuallyInstalledPackage
|   +-- config.json
|    +-- etc...
+-- AnotherPackage
|   +-- config.json
|   +-- etc...
```

Not

```
+ worlds
+-- YourManuallyInstalledPackage
|   +-- YourManuallyInstalledPackage
|       +-- config.json
|   +-- etc...
+-- AnotherPackage
|   +-- config.json
|   +-- etc...
```

1.5.4 Print Information

There are several convenience functions provided to allow packages, worlds, and scenarios to be easily inspected.

```
>>> packagemanager.package_info("Ocean")
Package: Ocean
   Platform: Linux
   Version: 0.1.0
   Path: LinuxNoEditor/Holodeck/Binaries/Linux/Holodeck
   Worlds:
   Rooms
         Scenarios:
         Rooms-DataGen:
            Agents:
               Name: turtle0
               Type: TurtleAgent
               Sensors:
                  LocationSensor
                     lcm_channel: POSITION
                  RotationSensor
                     lcm_channel: ROTATION
                  RangeFinderSensor
                     lcm_channel: LIDAR
                     configuration
                        LaserCount: 64
                        LaserMaxDistance: 20
                        LaserAngle: 0
                        LaserDebug: True
         Rooms-IEKF:
            Agents:
               Name: uav0
               Type: UavAgent
               Sensors:
                  PoseSensor
                  VelocitySensor
                  IMUSensor
   SimpleUnderwater
         Scenarios:
         SimpleUnderwater-AUV:
            Agents:
               Name: auv0
```

(continues on next page)

(continued from previous page)

```
            Type: HoveringAUV
            Sensors:
               PoseSensor
                  socket: IMUSocket
               VelocitySensor
                  socket: IMUSocket
               IMUSensor
                  socket: IMUSocket
               DVLSensor
                  socket: DVLSocket
```

You can also look for information for a specific world or scenario

```
packagemanager.world_info("SimpleUnderwater")
packagemanager.scenario_info("Rooms-DataGen")
```

CHAPTER

TWO

GETTING STARTED & EXAMPLES

First, see *Installation* to get the `holoocean` package and `Ocean` installed.

A minimal HoloOcean usage example is below:

```
import holoocean
import numpy as np

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

# The hovering AUV takes a command for each thruster
command = np.array([10,10,10,10,0,0,0,0])

for _ in range(180):
    state = env.step(command)
```

Notice that:

1. You pass the name of a *scenario* into `holoocean.make`

   See *Packages* for all of the different worlds and scenarios that are available.

2. The interface of HoloOcean is designed to be familiar to OpenAI Gym

You can access data from a specific sensor with the state dictionary:

```
dvl = state["DVLSensor"]
```

That's it! HoloOcean is meant to be fairly simple to use.

Check out the different *worlds* that are available, read the *API documentation*, or get started on making your own custom *scenarios*.

Below are some snippets that show how to use different aspects of HoloOcean.

2.1 Visualizing RGBCamera Output

It can be useful to display the output of the RGB camera while an agent is training. Below is an example using the `cv2` library.

When the window is open, press the `0` key to tick the environment and show the next window.

```
import holoocean, cv2

env = holoocean.make("Dam-HoveringCamera")
env.act('auv0', [10,10,10,10,0,0,0,0])

for _ in range(200):
    state = env.tick()

    if "LeftCamera" in state:
        pixels = state["LeftCamera"]
        cv2.namedWindow("Camera Output")
        cv2.imshow("Camera Output", pixels[:, :, 0:3])
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

2.2 Manually Controlling

We've found that *pynput* is a good library for sending keyboard commands to the agents for manual control.

Here's an example of controlling the *HoveringAUV* using the following keyboard shortcuts.

Key	Forward Key	Backward Key
Up/Down	i	k
Yaw Left/Right	j	l
Forward/Backward	w	s
Strafe Left/Right	a	d

```
import holoocean
import numpy as np
from pynput import keyboard

pressed_keys = list()
force = 25

def on_press(key):
    global pressed_keys
    if hasattr(key, 'char'):
        pressed_keys.append(key.char)
        pressed_keys = list(set(pressed_keys))

def on_release(key):
    global pressed_keys
    if hasattr(key, 'char'):
        pressed_keys.remove(key.char)

listener = keyboard.Listener(
    on_press=on_press,
    on_release=on_release)
listener.start()
```

(continues on next page)

(continued from previous page)

```
def parse_keys(keys, val):
    command = np.zeros(8)
    if 'i' in keys:
        command[0:4] += val
    if 'k' in keys:
        command[0:4] -= val
    if 'j' in keys:
        command[[4,7]] += val
        command[[5,6]] -= val
    if 'l' in keys:
        command[[4,7]] -= val
        command[[5,6]] += val

    if 'w' in keys:
        command[4:8] += val
    if 's' in keys:
        command[4:8] -= val
    if 'a' in keys:
        command[[4,6]] += val
        command[[5,7]] -= val
    if 'd' in keys:
        command[[4,6]] -= val
        command[[5,7]] += val

    return command

with holoocean.make("Dam-Hovering") as env:
    while True:
        if 'q' in pressed_keys:
            break
        command = parse_keys(pressed_keys, force)

        #send to holoocean
        env.act("auv0", command)
        state = env.tick()
```

2.3 Custom Scenario Configurations

HoloOcean worlds are meant to be configurable by changing out the scenario (see *Scenarios*). There are some scenarios included with HoloOcean packages distributed as `.json` files, but HoloOcean is intended to be used with user-created scenarios as well.

These can be created using a dictionary in a Python script or by creating a `.json` file. Both methods follow the same format, see *Scenario File Format*

2.3.1 Using a Dictionary for a Scenario Config

Create a dictionary in Python that matches the structure specified in *Scenario File Format*, and pass it in to `holoocean.make()`.

Example

```
import holoocean

cfg = {
    "name": "test_rgb_camera",
    "world": "SimpleUnderwater",
    "package_name": "Ocean",
    "main_agent": "auv0",
    "ticks_per_sec": 60,
    "agents": [
        {
            "agent_name": "auv0",
            "agent_type": "HoveringAUV",
            "sensors": [
                {
                    "sensor_type": "RGBCamera",
                    "socket": "CameraSocket",
                    "configuration": {
                        "CaptureWidth": 512,
                        "CaptureHeight": 512
                    }
                }
            ],
            "control_scheme": 0,
            "location": [0, 0, -10]
        }
    ]
}

with holoocean.make(scenario_cfg=cfg) as env:
    for _ in range(200):
        env.tick()
```

2.3.2 Using a `.json` file for a Scenario Config

You can specify a custom scenario by creating a `.json` file that follows the format given in *Scenario File Format* and either:

1. Placing it in HoloOcean's scenario search path
2. Loading it yourself and parsing it into a dictionary, and then using that dictionary as described in *Using a Dictionary for a Scenario Config*

HoloOcean's Scenario Search Path

When you give a scenario name to `holoocean.make()`, HoloOcean will search look each package folder (see *Package Installation Location*) until it finds a `.json` file that matches the scenario name.

So, you can place your custom scenario `.json` files in that folder and HoloOcean will automatically find and use it.

Warning: If you remove and re-install a package, HoloOcean will clear the contents of that folder

2.4 Multi Agent Example

With HoloOcean, you can control more than one agent at once. Instead of calling `.step()`, which both

1. passes a single command to the main agent, and
2. ticks the simulation

you should call `.act()`. Act supplies a command to a specific agent, but doesn't tick the game.

Once all agents have received their actions, you can call `.tick()` to tick the game.

After calling `.act()`, every time you call `.tick()` the same command will be supplied to the agent. To change the command, just call `.act()` again.

The state returned from tick is also somewhat different.

The state is now a dictionary from agent name to sensor dictionary.

Press tab to switch the viewport between agents. See *Hotkeys* for more.

```
import holoocean
import numpy as np

cfg = {
    "name": "test_rgb_camera",
    "world": "SimpleUnderwater",
    "package_name": "Ocean",
    "main_agent": "auv0",
    "ticks_per_sec": 60,
    "agents": [
        {
            "agent_name": "auv0",
            "agent_type": "TorpedoAUV",
            "sensors": [
                {
                    "sensor_type": "IMUSensor"
                }
            ],
            "control_scheme": 0,
            "location": [0, 0, -5]
        },
        {
            "agent_name": "auv1",
            "agent_type": "HoveringAUV",
            "sensors": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "sensor_type": "DVLSensor"
                }
            ],
            "control_scheme": 0,
            "location": [0, 2, -5]
        }
    ]
}

env = holoocean.make(scenario_cfg=cfg)
env.reset()

env.act('auv0', np.array([0,0,0,0,75]))
env.act('auv1', np.array([0,0,0,0,20,20,20,20]))
for i in range(300):
    states = env.tick()

    # states is a dictionary
    imu = states["auv0"]["IMUSensor"]

    vel = states["auv1"]["DVLSensor"]
```

2.5 Multi-Agent Communications

Many times it's necessary to communicate between agents. This can be done using the `AcousticBeaconSensor` or the `OpticalModemSensor`. Below are some examples of these

2.5.1 Sending Acoustic Messages

The command *holoocean.environments.HoloOceanEnvironment.send_acoustic_message()* is used to send messages between acoustic beacons. There's a number of message types that can be sent, all with varying functionality, see *holoocean.sensors.AcousticBeaconSensor* for details.

Further, a few helper functions exist if needed,

- *holoocean.environments.HoloOceanEnvironment.beacons* returns all beacons.
- *holoocean.environments.HoloOceanEnvironment.beacons_id* returns all beacons' ids.
- *holoocean.environments.HoloOceanEnvironment.beacons_status* returns all beacons' status (whether it's transmitting or not).

```
import holoocean

cfg = {
    "name": "test_acou_coms",
    "world": "SimpleUnderwater",
    "package_name": "Ocean",
    "main_agent": "auv0",
    "ticks_per_sec": 200,
```

(continues on next page)

(continued from previous page)

```
    "agents": [
        {
            "agent_name": "auv0",
            "agent_type": "HoveringAUV",
            "sensors": [
                {
                    "sensor_type": "AcousticBeaconSensor",
                    "location": [0,0,0],
                    "configuration": {
                        "id": 0
                    }
                },
            ],
            "control_scheme": 0,
            "location": [0, 0, -5]
        },
        {
            "agent_name": "auv1",
            "agent_type": "HoveringAUV",
            "sensors": [
                {
                    "sensor_type": "AcousticBeaconSensor",
                    "location": [0,0,0],
                    "configuration": {
                        "id": 1
                    }
                },
            ],
            "control_scheme": 0,
            "location": [0, 100, -5]
        }
    ]
}

env = holoocean.make(scenario_cfg=cfg)
env.reset()

# This is how you send a message from one acoustic com to another
# This sends from id 0 to id 1 (ids configured above)
# with message type "OWAY" and data "my_data_payload"
env.send_acoustic_message(0, 1, "OWAY", "my_data_payload")

for i in range(300):
    states = env.tick()
    if "AcousticBeaconSensor" in states['auv1']:
        # For this message, should receive back [message_type, from_sensor, data_payload]
        print(i, "Received:", states['auv1']["AcousticBeaconSensor"])
        break
    else:
        print(i, "No message received")
```

2.5.2 Sending Optical Messages

The command *holoocean.environments.HoloOceanEnvironment.send_optical_message()* is used to send messages between optical modems. See *holoocean.sensors.OpticalModemSensor* for configuration details. Note in order for a message to be transmitted, the 2 sensors must be aligned.

Further, a few helper functions exist if needed,

- *holoocean.environments.HoloOceanEnvironment.modems* returns all modems.
- *holoocean.environments.HoloOceanEnvironment.modems_id* returns all modems' ids.

```
import holoocean

cfg = {
    "name": "test_acou_coms",
    "world": "SimpleUnderwater",
    "package_name": "Ocean",
    "main_agent": "auv0",
    "ticks_per_sec": 200,
    "agents": [
        {
            "agent_name": "auv0",
            "agent_type": "HoveringAUV",
            "sensors": [
                {
                    "sensor_type": "OpticalModemSensor",
                    "location": [0,0,0],
                    "socket": "SonarSocket",
                    "configuration": {
                        "id": 0
                    }
                },
            ],
            "control_scheme": 0,
            "location": [25, 0, -5],
            "rotation": [0, 0, 180]
        },
        {
            "agent_name": "auv1",
            "agent_type": "HoveringAUV",
            "sensors": [
                {
                    "sensor_type": "OpticalModemSensor",
                    "location": [0,0,0],
                    "socket": "SonarSocket",
                    "configuration": {
                        "id": 1
                    }
                },
            ],
            "control_scheme": 0,
            "location": [0, 0, -5]
        }
    ]
```

(continues on next page)

(continued from previous page)

```
}

env = holoocean.make(scenario_cfg=cfg)
env.reset()

# This is how you send a message from one optical com to another
# This sends from id 0 to id 1 (ids configured above)
# with data "my_data_payload"
env.send_optical_message(0, 1, "my_data_payload")

for i in range(300):
    states = env.tick()
    if "OpticalModemSensor" in states['auv1']:
        print(i, "Received:", states['auv1']["OpticalModemSensor"])
        break
    else:
        print(i, "No message received")
```

2.6 Visualizing Imaging Sonar

It can be useful to visualize the output of the sonar sensor during a simulation. This script will do that, plotting each time sonar data is received.

Note, running this script will create octrees while being ran and may cause some pauses. See *Octree Generation* for workarounds and more info.

```
import holoocean
import matplotlib.pyplot as plt
import numpy as np

#### GET SONAR CONFIG
scenario = "PierHarbor-HoveringImagingSonar"
config = holoocean.packagemanager.get_scenario(scenario)
config = config['agents'][0]['sensors'][-1]["configuration"]
azi = config['Azimuth']
minR = config['RangeMin']
maxR = config['RangeMax']
binsR = config['RangeBins']
binsA = config['AzimuthBins']

#### GET PLOT READY
plt.ion()
fig, ax = plt.subplots(subplot_kw=dict(projection='polar'), figsize=(8,5))
ax.set_theta_zero_location("N")
ax.set_thetamin(-azi/2)
ax.set_thetamax(azi/2)

theta = np.linspace(-azi/2, azi/2, binsA)*np.pi/180
r = np.linspace(minR, maxR, binsR)
T, R = np.meshgrid(theta, r)
```

(continues on next page)

(continued from previous page)

```
z = np.zeros_like(T)

plt.grid(False)
plot = ax.pcolormesh(T, R, z, cmap='gray', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
fig.canvas.draw()
fig.canvas.flush_events()

#### RUN SIMULATION
command = np.array([0,0,0,0,-20,-20,-20,-20])
with holoocean.make(scenario) as env:
    for i in range(1000):
        env.act("auv0", command)
        state = env.tick()

        if 'ImagingSonar' in state:
            s = state['ImagingSonar']
            plot.set_array(s.ravel())

            fig.canvas.draw()
            fig.canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

2.7 Visualizing Profiling Sonar

It can be useful to visualize the output of the sonar sensor during a simulation. This script will do that, plotting each time sonar data is received.

Note, running this script will create octrees while being ran and may cause some pauses. See *Octree Generation* for workarounds and more info.

```
import holoocean
import matplotlib.pyplot as plt
import numpy as np

#### GET SONAR CONFIG
scenario = "OpenWater-TorpedoProfilingSonar"
config = holoocean.packagemanager.get_scenario(scenario)
config = config['agents'][0]['sensors'][-1]["configuration"]
azi = config['Azimuth']
minR = config['RangeMin']
maxR = config['RangeMax']
binsR = config['RangeBins']
binsA = config['AzimuthBins']

#### GET PLOT READY
plt.ion()
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(subplot_kw=dict(projection='polar'), figsize=(8,5))
ax.set_theta_zero_location("N")
ax.set_thetamin(-azi/2)
ax.set_thetamax(azi/2)

theta = np.linspace(-azi/2, azi/2, binsA)*np.pi/180
r = np.linspace(minR, maxR, binsR)
T, R = np.meshgrid(theta, r)
z = np.zeros_like(T)

plt.grid(False)
plot = ax.pcolormesh(T, R, z, cmap='gray', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
fig.canvas.flush_events()

#### RUN SIMULATION
command = np.array([0,0,0,0,20])
with holoocean.make(scenario) as env:
    for i in range(1000):
        env.act("auv0", command)
        state = env.tick()

        if 'ProfilingSonar' in state:
            s = state['ProfilingSonar']
            plot.set_array(s.ravel())

            fig.canvas.draw()
            fig.canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

2.8 Visualizing Sidescan Sonar

It can be useful to visualize the output of the sonar sensor during a simulation. This script will do that, plotting each time sonar data is received.

Note, running this script will create octrees while being ran and may cause some pauses. See *Octree Generation* for workarounds and more info.

```
import holoocean
import matplotlib.pyplot as plt
import numpy as np

#### GET SONAR CONFIG
scenario = "OpenWater-TorpedoSidescanSonar"
config = holoocean.packagemanager.get_scenario(scenario)
config = config['agents'][0]['sensors'][-1]["configuration"]
maxR = config['RangeMax']
```

(continues on next page)

(continued from previous page)

```
binsR = config['RangeBins']

#### GET PLOT READY
plt.ion()

t = np.arange(0,50)
r = np.linspace(-maxR, maxR, binsR)
R, T = np.meshgrid(r, t)
data = np.zeros_like(R)

plt.grid(False)
plot = plt.pcolormesh(R, T, data, cmap='copper', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
plt.gca().invert_yaxis()
plt.gcf().canvas.flush_events()

#### RUN SIMULATION
command = np.array([0,0,0,0,20])
with holoocean.make(scenario) as env:
    for i in range(1000):
        env.act("auv0", command)
        state = env.tick()

        if 'SidescanSonar' in state:
            data = np.roll(data, 1, axis=0)
            data[0] = state['SidescanSonar']

            plot.set_array(data.ravel())

            plt.draw()
            plt.gcf().canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

2.9 Visualizing Singlebeam Sonar

It can be useful to visualize the output of the sonar sensor during a simulation. This script will do that, plotting each time sonar data is received.

Note, running this script will create octrees while being ran and may cause some pauses. See *Octree Generation* for workarounds and more info.

```
import holoocean
import matplotlib.pyplot as plt
import numpy as np

#### GET SONAR CONFIG
scenario = "OpenWater-TorpedoSinglebeamSonar"
```

(continues on next page)

(continued from previous page)

```
config = holoocean.packagemanager.get_scenario(scenario)
config = config['agents'][0]['sensors'][-1]["configuration"]
minR = config['RangeMin']
maxR = config['RangeMax']
binsR = config['RangeBins']

#### GET PLOT READY
plt.ion()

t = np.arange(0,50)
r = np.linspace(minR, maxR, binsR)
T, R = np.meshgrid(t, r)
data = np.zeros_like(R)

plt.grid(False)
plot = plt.pcolormesh(T, R, data, cmap='gray', shading='auto', vmin=0, vmax=1)
plt.tight_layout()
plt.gca().invert_yaxis()
plt.gcf().canvas.flush_events()

#### RUN SIMULATION
command = np.array([0,0,0,0,20])
with holoocean.make(scenario) as env:
    for i in range(1000):
        env.act("auv0", command)
        state = env.tick()

        if 'SinglebeamSonar' in state:
            data = np.roll(data, 1, axis=1)
            data[:,0] = state['SinglebeamSonar']

            plot.set_array(data.ravel())

            plt.draw()
            plt.gcf().canvas.flush_events()

print("Finished Simulation!")
plt.ioff()
plt.show()
```

There is also an examples.py in the root of the holoocean repo with more example code.

CHAPTER

THREE

USING HOLOOCEAN

3.1 Scenarios

3.1.1 What is a scenario?

A scenario tells HoloOcean which world to load, which agents to place in the world, and which sensors they need.

It defines:

- Which world to load
- Agent Definitions
  - What type of agent they are
  - Where they are
  - What sensors they have

Tip: You can think of scenarios like a map or gametype variant from Halo: the world or map itself doesn't change, but the things in the world and your objective can change.

Scenarios allow the same world to be used for many different purposes, and allows you to extend and customize the scenarios we provide to suit your needs without repackaging the engine.

When you call `holoocean.make()` to create an environment, you pass in the name of a scenario, eg `holoocean.make("Pier-Hovering")`. This tells HoloOcean which world to load and where to place agents.

3.1.2 Scenario File Format

Scenario `.json` files are distributed in packages (see *Package Contents*), and must be named `{WorldName}-{ScenarioName}.json`. By default they are stored in the `worlds/{PackageName}` directory, but they can be loaded from a Python dictionary as well.

Scenario File

```
{
   "name": "{Scenario Name}",
   "world": "{world it is associated with}",
   "lcm_provider": "{Optional, where to publish lcm to}",
   "ticks_per_sec": 30,
   "frames_per_sec": 30,
   "env_min": [-10, -10, -10],
   "env_max": [10, 10, 10],
   "octree_min": 0.1,
   "octree_max": 5,
   "agents":[
      "array of agent objects"
   ],
   "weather": {
      "hour": 12,
      "type": "'sunny' or 'cloudy' or 'rain'",
      "fog_density": 0,
      "day_cycle_length": 86400
   },
   "window_width":  1280,
   "window_height": 720
}
```

`window_width/height` control the size of the window opened when an environment is created. For more information about weather options, see weather.

Note: The first agent in the `agents` array is the "main agent"

Frame Rates

There's two parameters you can configure that'll handle frame rate changes: `ticks_per_sec` and `frames_per_sec`.

`ticks_per_sec` changes how many ticks in a simulation second. This must be higher than any "Hz" sampling rate of the sensors used. Defaults to 30.

`frames_per_sec` is the max FPS the environment can run at. If *true*, it will match `ticks_per_sec`. If *false*, FPS will not be capped, and the environment will run as fast as possible. If a number, that'll be the frame rate cap.

For a few examples of how you might want to configure these. If you're manually controlling the robot(s), you'll likely want it to run at realtime, thus you'll want to set `frames_per_sec` to true. When using a quality GPU, simulations can run much faster than realtime, making things difficult to control otherwise. If you're running headless/autonomous, you'll likely want the simulation to run as fast as possible, thus a good `frames_per_sec` would be false.

Configuring Octree

When using a form of sonar sensor and initializing the world, an Octree will either be created or loaded from a cache. The parameters of these can be set using the `env_min`, `env_max`, `octree_min`, and `octree_max`. The octrees are cached in the `LinuxNoEditor/Holodeck/Octrees` folder in the worlds folder. See *Package Installation Location*.

`env_min`/`env_max` are used to set the upper/lower bounds of the environment. They should be set in *Package Structure*, but the values set here will override it.

`octree_min`/`octree_max` are used to set the minimum/mid-level size of the octree. `octree_min` can go as low as .01 (1cm), and then the octree will double in size till it reaches `octree_max`.

Agent objects

```
{
   "agent_name": "uav0",
   "agent_type": "{agent types}",
   "sensors": [
      "array of sensor objects"
   ],
   "control_scheme": "{control scheme type}",
   "location": [1.0, 2.0, 3.0],
   "rotation": [1.0, 2.0, 3.0],
   "location_randomization": [1, 2, 3],
   "rotation_randomization": [10, 10, 10]
}
```

Note: HoloOcean coordinates are **right handed** in meters. See *Coordinate System*

Location Randomization

`location_randomization` and `rotation_randomization` are optional. If provided, the agent's start location and/or rotation will vary by a random amount between the negative and the positive values of the provided randomization values as sampled from a uniform distribution.

The location value is measured in meters, in the format `[dx, dy, dz]` and the rotation is `[roll, pitch, yaw]`, rotated about XYZ fixed axes, ie R_z R_y R_x.

Agent Types

Here are valid `agent_type` s:

Agent Type	String in agent_type
HoveringAUV	`HoveringAUV`
TorpedoAUV	`TorpedoAUV`
TurtleAgent	`TurtleAgent`
UavAgent	`UAV`

Control Schemes

Control schemes are represented as an integer. For valid values and a description of how each scheme works, see the documentation pages for each agent.

Sensor Objects

```
{
    "sensor_type": "RGBCamera",
    "sensor_name": "FrontCamera",
    "location": [1.0, 2.0, 3.0],
    "rotation": [1.0, 2.0, 3.0],
    "socket": "socket name or \"\"",
    "Hz": 5,
    "lcm_channel": "channel_name",
    "configuration": {

    }
}
```

Sensors have a couple options for placement.

1. **Provide a socket name**

   This will place the sensor in the given socket

   ```
   {
       "sensor_type": "RGBCamera",
       "socket": "CameraSocket"
   }
   ```

2. **Provide a socket, location and/or rotation**

   The sensor will be placed offset to the socket by the location and rotation. The rotation is `[roll, pitch, yaw]` in degrees, rotated about XYZ fixed axes, ie R_z R_y R_x.

   ```
   {
       "sensor_type": "RGBCamera",
       "location": [1.0, 2.0, 3.0],
       "rotation": [1.0, 2.0, 3.0],
       "socket": "CameraSocket"
   }
   ```

3. **Provide just a location and/or rotation**

   The sensor will be placed at the given coordinates, offset from the root of the agent.

   ```
   {
       "sensor_type": "RGBCamera",
       "location": [1.0, 2.0, 3.0],
       "rotation": [1.0, 2.0, 3.0]
   }
   ```

4. **Provide a sensor sample rate**

The sensor will be sampled at this rate. Note this must be less then `ticks_per_sec`, and preferably a divisor of `ticks_per_sec` as well. See *Frame Rates* for more info on `ticks_per_sec`.

```
{
    "sensor_type": "RGBCamera",
    "Hz": 20
}
```

5. **Publish Message**

   Currently, HoloOcean supports publishing mesages to LCM (with possible ROS package coming). To publish sensor data to LCM, specify the type to publish.

   ```
   {
       "sensor_type": "RGBCamera",
       "lcm_channel": "CAMERA"
   }
   ```

   The channel parameter specifies which channel to publish the sensor data to.

The only keys that are required in a sensor object is `"sensor_type"`, the rest will default as shown below

```
{
    "sensor_name": "sensor_type",
    "location": [0, 0, 0],
    "rotation": [0, 0, 0],
    "socket": "",
    "publish": "",
    "lcm_channel": "",
    "configuration": {}
}
```

Configuration Block

The contents of the `configuration` block are sensor-specific. That block is passed verbatim to the sensor itself, which parses it.

For example, the docstring for *RGBCamera* states that it accepts `CaptureWidth` and `CaptureHeight` parameters, so an example sensor configuration would be:

```
{
    "sensor_name": "RBGCamera",
    "socket": "CameraSocket",
    "configuration": {
        "CaptureHeight": 1920,
        "CaptureWidth": 1080
    }
}
```

3.2 Viewport Hotkeys

When the viewport window is open, and the environment is being ticked (with calls to `tick()` or `step()`, there are a few hotkeys you can use.

3.2.1 Hotkeys

The AgentFollower, or the camera that the viewport displays, can be manipulated as follows:

Key	Action	Description
`c`	Toggle camera mode	Toggles the camera from a chase camera and perspective camera, which shows what the agent's camera sensor sees.
`v`	Toggle spectator mode	Toggles spectator mode, which allows you to free-cam around the world.
`w a s d`	Move camera	Move the viewport camera around when in spectator/free-cam mode.
`q ctrl`	Descend	For spectator/free-cam mode
`e space`	Ascend	For spectator/free-cam mode
`shift`	Turbo	Move faster when in spectator/free-cam
`tab`	Cycle through agents	When not in spectator/free-cam mode, cycles through the agents in the world
`h`	Toggle HUD	The HUD displays the name and location of the agent the viewport is following, or the location of the camera if the viewport is detached (spectator mode) Note that this will interfere with the ViewportCapture sensor

Opening Console

Pressing ~ will open Unreal Engine 4's developer console, which has a few useful commands. See the Unreal Docs for a complete list of commands.

Useful Commands

- `stat fps`

  Prints the frames per second on the screen.

3.3 Publishing Sensor Data

Currently, HoloOcean supports publishing data to LCM (with a potential ROS wrapper being considered). All this config happens in the *scenario* file. We'll outline what this takes here.

LCM publishes data to a certain medium, called the provider. This can be locally, over the network, a log file, etc. This can be specified in the header of the scenario file. See _here for options on this. If no provider is specified, HoloOcean uses the default LCM udqm.

```
{
    "name": "{Scenario Name}",
    "world": "{world it is associated with}",
    "lcm_provider": "file:///home/lcm.log"
    "agents":[
        "array of agent objects"
    ]
}
```

Once the provider is chosen, HoloOcean publishes each sensor individually. The lcm_channel is then chosen by the sensor config. If no channel is specified, the sensor will not be published.

```
{
    "sensor_type": "RGBCamera",
    "sensor_name": "FrontCamera",
    "lcm_channel": "CAMERA",
}
```

3.4 Units and Coordinates in HoloOcean

HoloOcean uses **meters** for units and a **right-handed coordinate system** for all locations, distances, and offsets.

3.4.1 Coordinate System

Unreal Engine uses a left handed coordinate system by default, however to keep with general robotics conventions, we use a right handed coordinate system with positive `z` being up.

So, when you need to specify a location in HoloOcean, the format is as follows

`[x, y, z]` where:

- Positive `x` is **forward**
- Positive `y` is **left**
- Positive `z` is **up**

Remember that the units for `[x, y, z]` are in meters (Unreal Engine defaults to centimeters, we've changed this to make things a bit easier).

3.4.2 Rotations

Rotations are specified in `[roll, pitch, yaw]` / `[x, y, z]` format, in in degrees (usually). This means

- **Roll**: Rotation around the fixed forward (`x`) axis
- **Pitch**: Rotation around the fixed right (`y`) axis
- **Yaw**: Rotation around the fixed up (`z`) axis

In that order.

3.5 Improving HoloOcean Performance

HoloOcean is fairly performant by default, but you can also sacrifice features to increase your frames per second.

3.5.1 RGBCamera

By far, the biggest single thing you can do to improve performance is to disable the `RGBCamera`. Rendering the camera every frame causes a context switch deep in the rendering code of the engine, which has a significant performance penalty.

This chart shows how much performance you can expect to gain or loose adjusting the RGBCamera (left column is frame time in milleseconds). Note all these tests were done in the original Holodeck, but the results should be the same.

Resolution	UrbanCity		MazeWorld		AndroidPlayground	
No Camera	8.55 ms	117 fps	4.69 ms	213 fps	2.47 ms	405 fps
64	17 ms	59 fps	11 ms	91 fps	4.87 ms	205 fps
128	20 ms	50 fps	11.6 ms	86 fps	5.59 ms	179 fps
256	22 ms	45 fps	14.71 ms	68 fps	9.02 ms	111 fps
512	35 ms	29 fps	30.8 ms	32 fps	24.81 ms	40 fps
1024	89 ms	11 fps	84.2 ms	12 fps	94.55 ms	11 fps
2048	410 ms	2 fps	383 ms	3 fps	366 ms	3 fps

Disabling the `RGBCamera`

Remove the `RGBCamera` entry from the scenario configuration file you are using.

See *Custom Scenario Configurations*.

Lowering the RGBCamera resolution

Lowering the resolution of the RGBCamera can also help speed things up. Create a *custom scenario* and in the *configuration block* for the RGBCamera set the CaptureWidth and CaptureHeight.

See *RGBCamera* for more details.

Changing ticks per capture

The camera sample rate can be reduced to increase the average frames per second. See *Sensor Objects* and the Hz parameter for more info.

3.5.2 Sonar Sensors

The sonar sensors can also be taxing on performance. There's a number of things that can be done to help improve their performance as well.

Lowering Octree Resolution

The Octree resolution has a large impact on sonar performance. The higher octree_min is, the less leaves there are to search through, and the faster it'll run. This will have an impact on image quality, especially at close distances. If most objects that are being inspected are a ways away, this parameter can be safely increased quite a bit.

See *Configuring Octree* for info on how to do that.

Changing ticks per capture

The sonar sample rate can be reduced to increase the average frames per second. See *Sensor Objects* and the Hz parameter for more info.

3.5.3 Disable Viewport Rendering

Rendering the viewport window can be unnecessary during training. You can disable the viewport with the *should_render_viewport()* method.

At lower RGBCamera resolutions, you can expect a ~40% frame time reduction.

3.5.4 Change Render Quality

You can adjust HoloOcean to render at a lower (or higher) quality to improve performance. See the *set_render_quality()* method

Below is a comparison of render qualities and the frame time in ms

Quality	MazeWorld	UrbanCity	AndroidPlayground
0	10.34	12.33	6.63
1	10.53	15.06	6.84
2	14.81	19.19	8.66
3	15.58	21.78	9.2

3.6 Using HoloOcean Headless

On Linux, HoloOcean can run headless without opening a viewport window. This can happen automatically, or you can force it to not appear

3.6.1 Headless Mode vs Disabling Viewport Rendering

These are two different features.

Disabling Viewport Rendering is calling the (*should_render_viewport()*) method on a *HoloOceanEnvironment*. This can be done at runtime. It will appear as if the image being rendered in the viewport has frozen, but *RGBCamera* s and other sensors will still update correctly.

Headless Mode is when the viewport window does not appear. If Headless Mode is manually enabled, it will also disable viewport rendering automatically.

3.6.2 Forcing Headless Mode

In `holoocean.make()`, set `show_viewport` to `False`.

Note: This will also disable viewport rendering (*should_render_viewport()*)

If you still want to render the viewport (ie for the *ViewportCapture*) when running headless, simply set (*should_render_viewport()*) to `True`

3.6.3 Automatic Headless Mode

If the engine does not detect the `DISPLAY` environment variable, it will not open a window. This will happen automatically if HoloOcean is run from a SSH session.

Note: This will not disable viewport rendering.

3.7 Octree Generation

When running an environment with a sonar sensor, octrees must be generated. These can often take up several gigabytes of storage, thus aren't feasible as part of the downloaded package.

Upon startup, all octrees within the `InitOctreeRange` parameter are created, then more are made as the agent moves throughout the environment. This can cause pauses in the simulation the first time it is ran. A warning will appear onscreen about this and can be disabled with the `ShowWarning` parameter. All subsequent simulation runs will use the cached octrees and run much faster.

One option to avoid waiting times, is to run the simulation without the viewport, and let it generate octrees in the background. Here's an example script that does exactly that, just change the scenario to the one that you would like to create octrees for,

```
import holoocean
import numpy as np
from tqdm import tqdm

command = np.array([0,0,0,0,-20,-20,-20,-20])
print("Building octrees...")
with holoocean.make("PierHarbor-HoveringImagingSonar", show_viewport=False) as env:
    for i in tqdm(range(1000)):
        env.act("auv0", command)
        state = env.tick()

print("Finished Simulation")
```

When octrees are made, they are saved in the *Package Installation Location* in an octree folder. This will be as follows,

Platform	Octree Save Location
Linux	`~/.local/share/holoocean/{holoocean_version}/worlds/{world_name}/LinuxNoEditor/Holodeck/Octrees`
Windows	`%USERPROFILE%\AppData\Local\holoocean\{holoocean_version}\worlds\{world_name}\WindowsNoEditor\Holodeck\Octrees`

In this octree folder, there will be additional folders for each level name, and in those a folder for each octree size used. If files are being actively saved here it means that the simulation is still running and isn't frozen.

CHAPTER

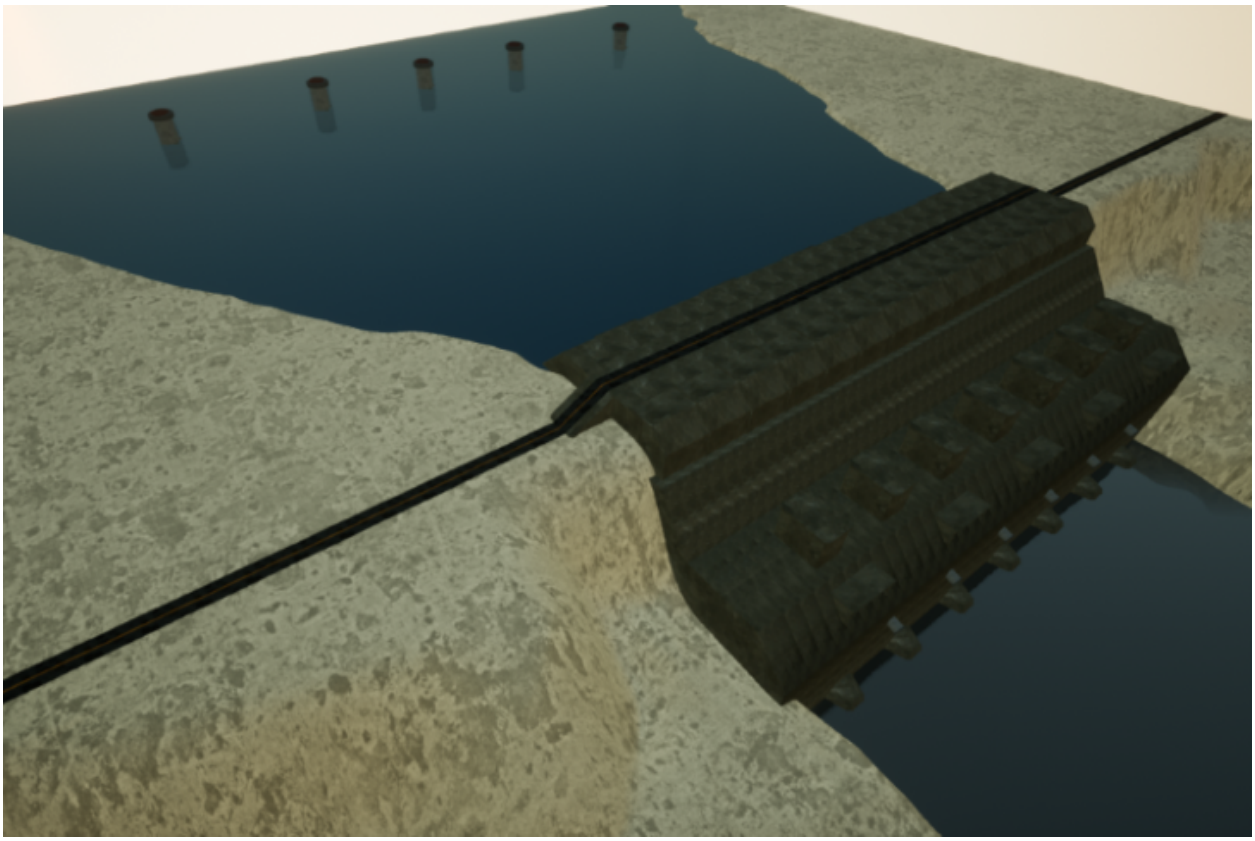
FOUR

HOLOOCEAN PACKAGES

These are the different packages available for download. A holoocean package contains one or more worlds, which each have one or more scenarios.

4.1 Ocean Package

4.1.1 Dam

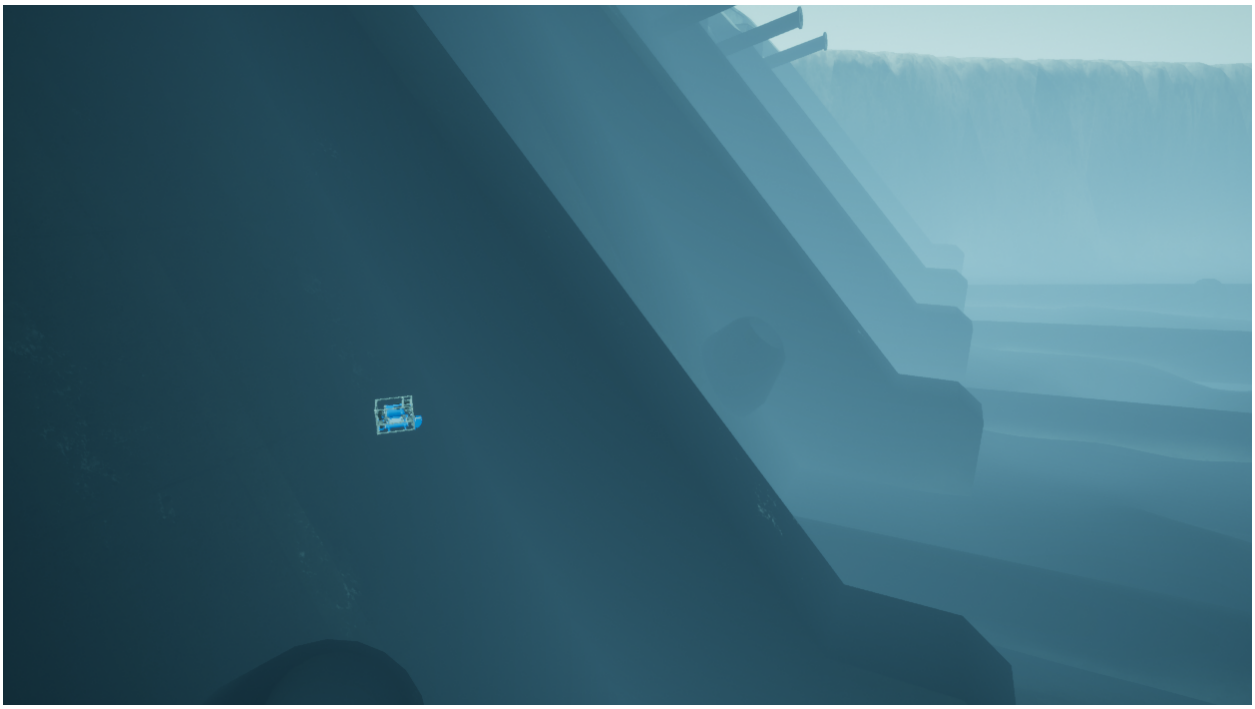


This is a dam that can be used for inspection. Entire environment is around 650m x 650m. There's a number of pipes that can be inspected as well as the dam itself.

Dam-Hovering

This scenario starts with a HoveringAUV near the actual dam. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

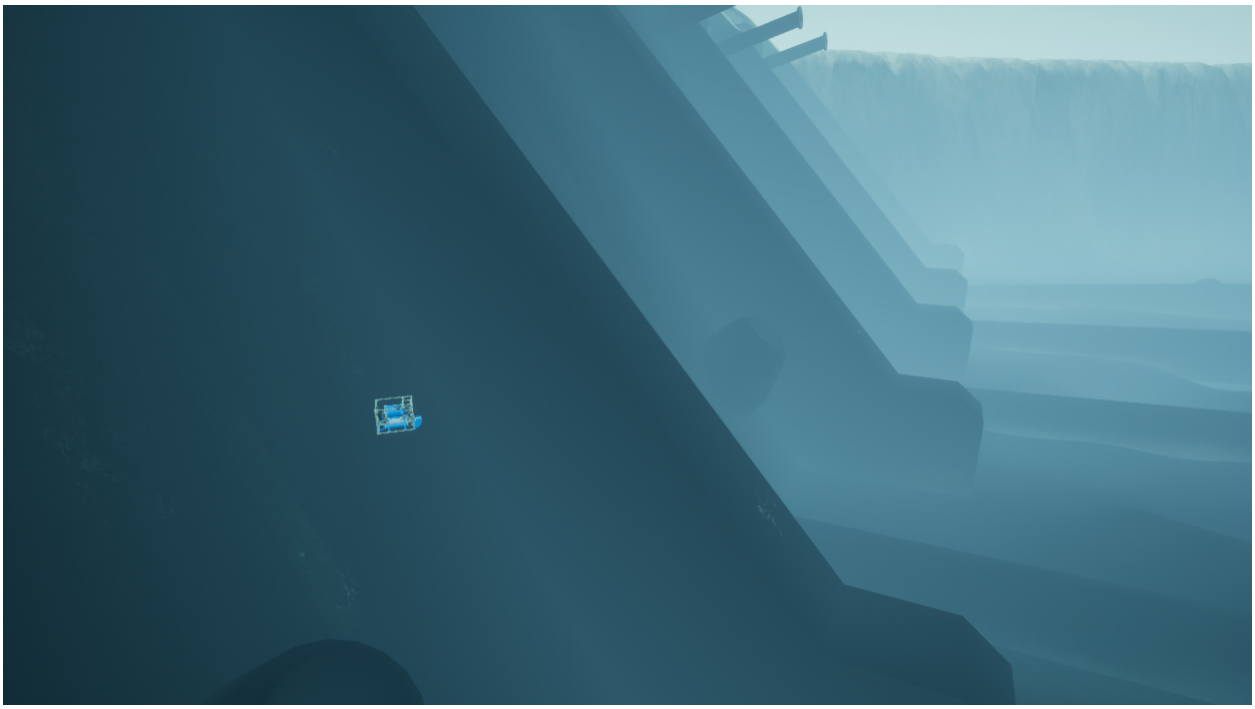


Dam-HoveringCamera

This scenario starts with a HoveringAUV near the actual dam. This is identical to the base version, only with cameras mounted. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *RGBCamera* named LeftCamera.
  - *RGBCamera* named RightCamera.

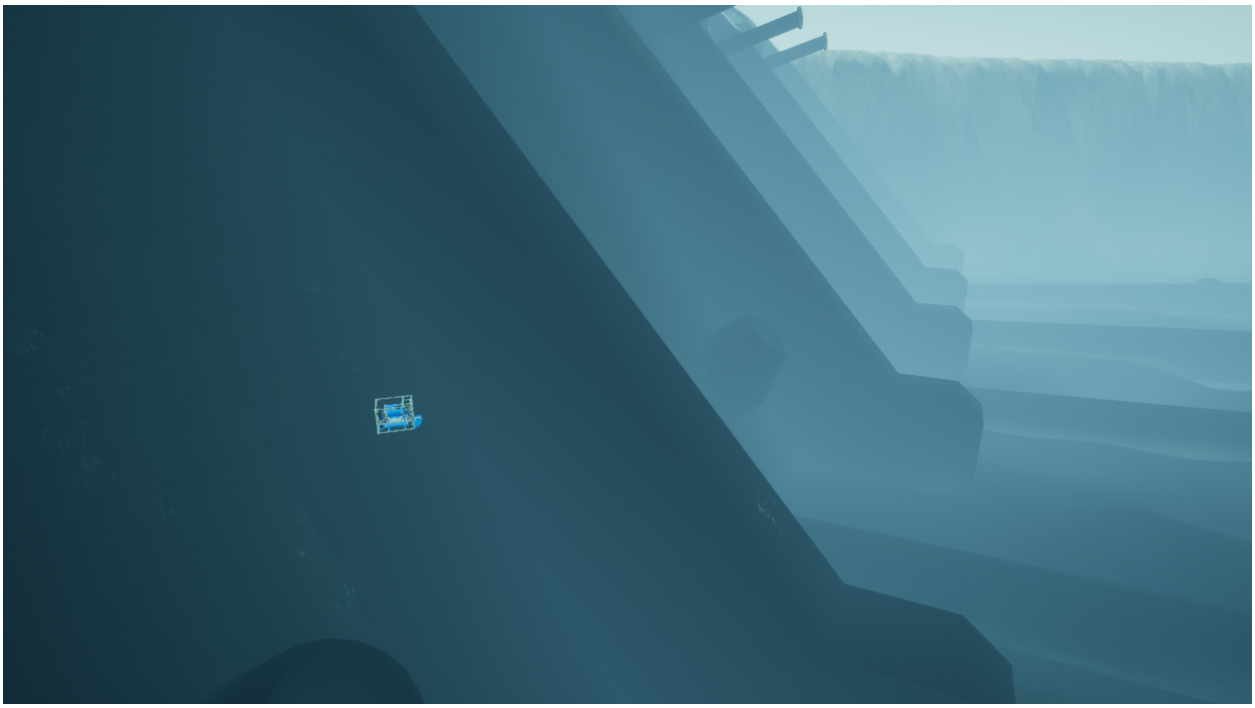
- *DepthSensor* configured with noise.
- *PoseSensor* for ground truth.
- *VelocitySensor* for ground truth.

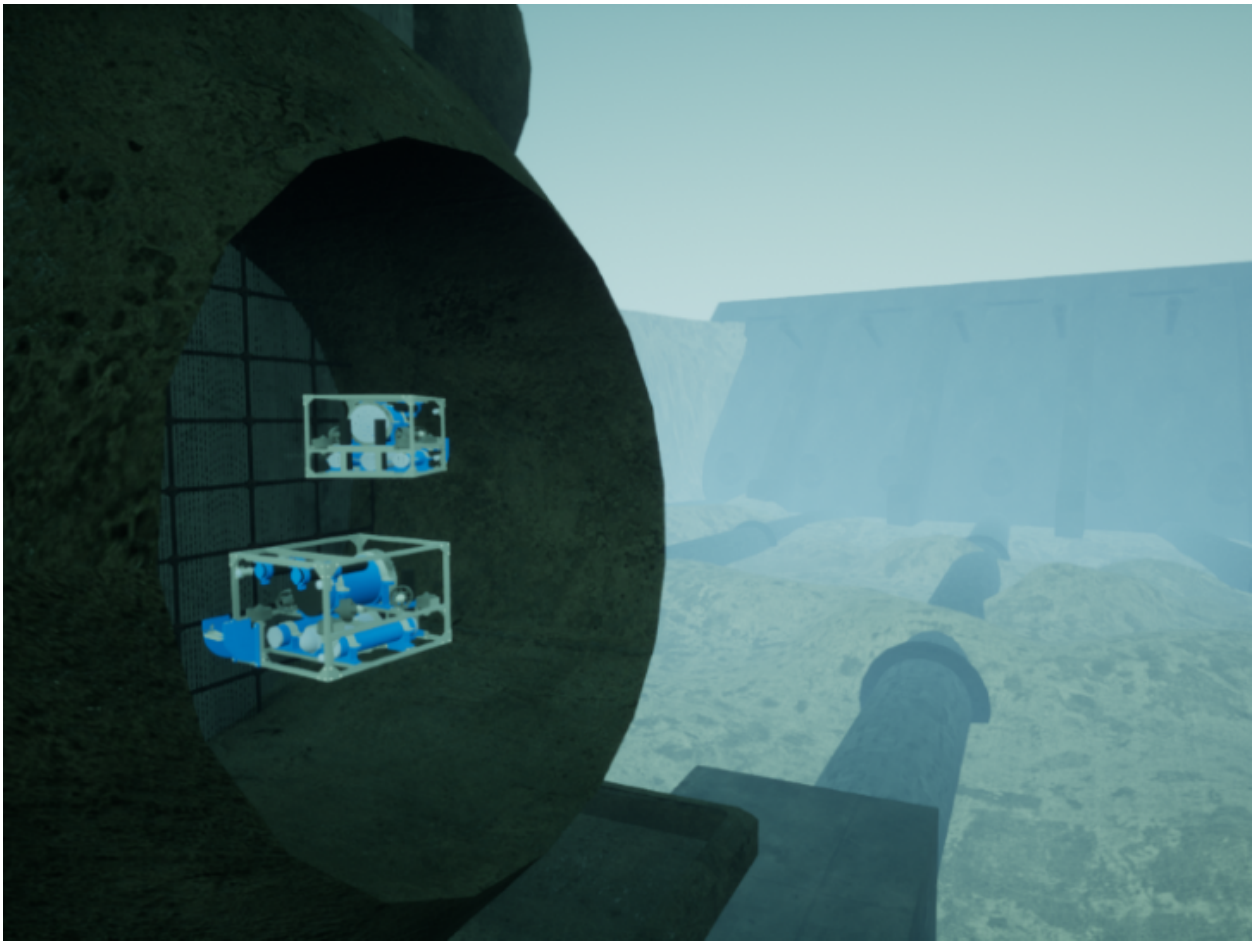


Dam-HoveringImagingSonar

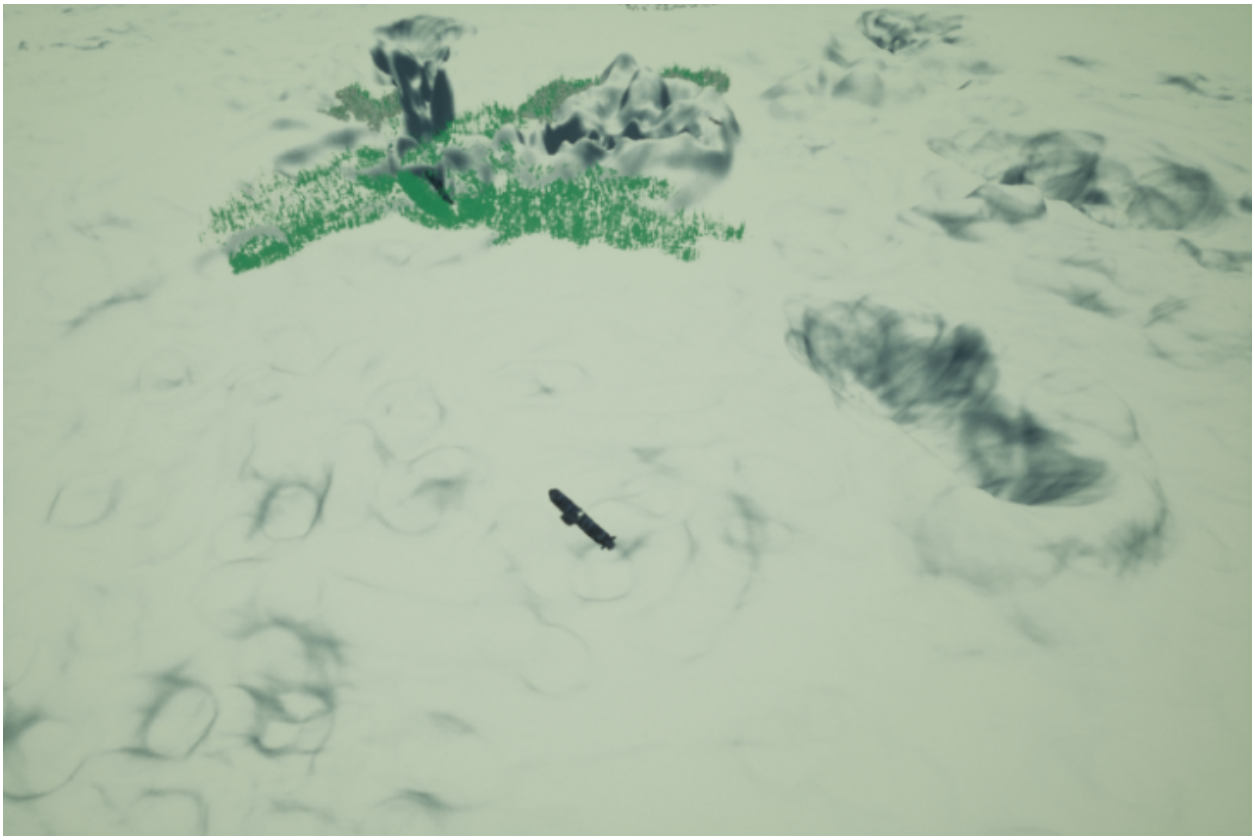
This scenario starts with a HoveringAUV near the actual dam. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *ImagingSonar* configured with an elevation of 20 degrees, azimuth 130, range 1-40m, an initial octree generation of 40m, and all noise turned on.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.





4.1.2 OpenWater



This is a large open water environment around 2km x 2km in size. There's a number of rolling hills, sunken submarines and planes, and plant life that can be inspected.

OpenWater-Hovering

This scenario starts with a HoveringAUV near a submarine. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

OpenWater-HoveringCamera

This scenario starts with a HoveringAUV near a submarine. This is identical to the base version, only with cameras mounted. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *RGBCamera* named LeftCamera.
  - *RGBCamera* named RightCamera.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

OpenWater-HoveringImagingSonar

This scenario starts with a HoveringAUV near a submarine. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *ImagingSonar* configured with an elevation of 20 degrees, azimuth 130, range 1-40m, an initial octree generation of 40m, and all noise turned on.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

OpenWater-Torpedo

This scenario starts with a TorpedoAUV near a submarine. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

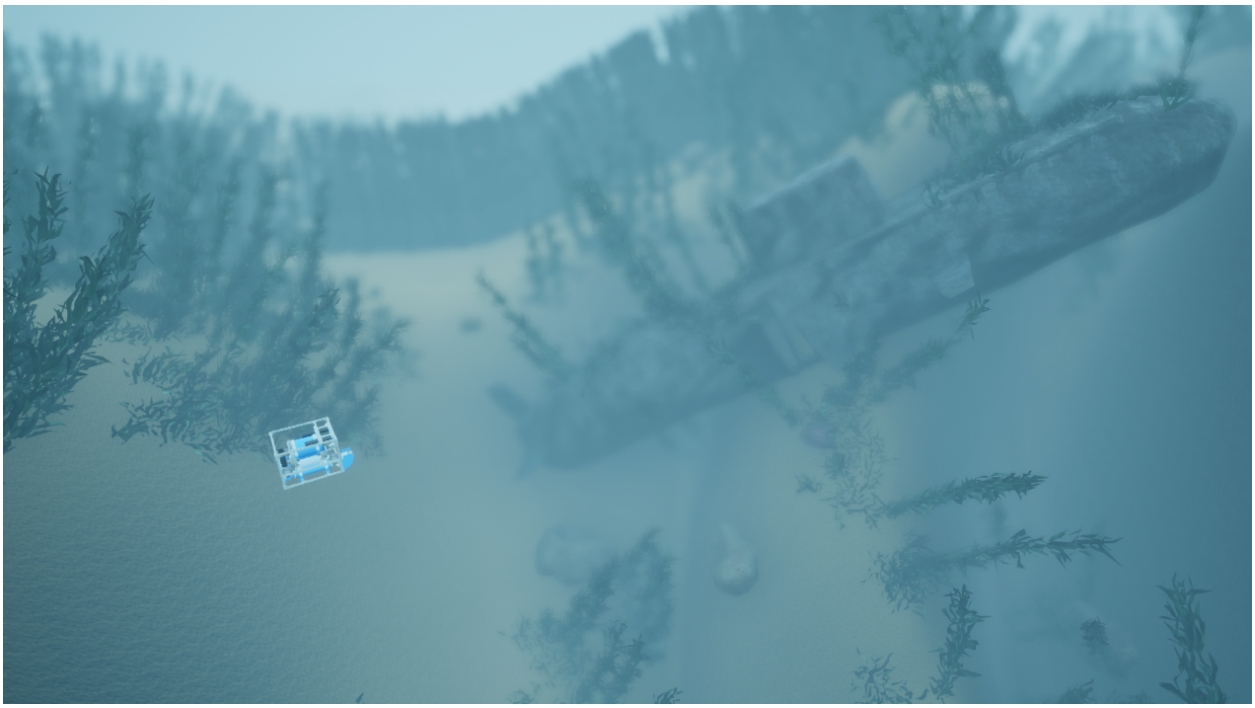
- **auv0: Main *HoveringAUV* agent**
  - `IMUSensor` configured with noise, bias, and returns bias.
  - `GPSSensor` gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - `DVLSensor` configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - `DepthSensor` configured with noise.
  - `PoseSensor` for ground truth.
  - `VelocitySensor` for ground truth.

OpenWater-TorpedoProfilingSonar

This scenario starts with a TorpedoAUV near a submarine. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *TorpedoAUV* agent**
  - `IMUSensor` configured with noise, bias, and returns bias.
  - `GPSSensor` gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - `DVLSensor` configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - `ProfilingSonar` configured with an elevation of 1 degrees, azimuth 120, range 1-60m, an initial octree generation of 60m, and all noise turned on.

- *DepthSensor* configured with noise.
- *PoseSensor* for ground truth.
- *VelocitySensor* for ground truth.



OpenWater-TorpedoSidescanSonar

This scenario starts with a TorpedoAUV near a submarine. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

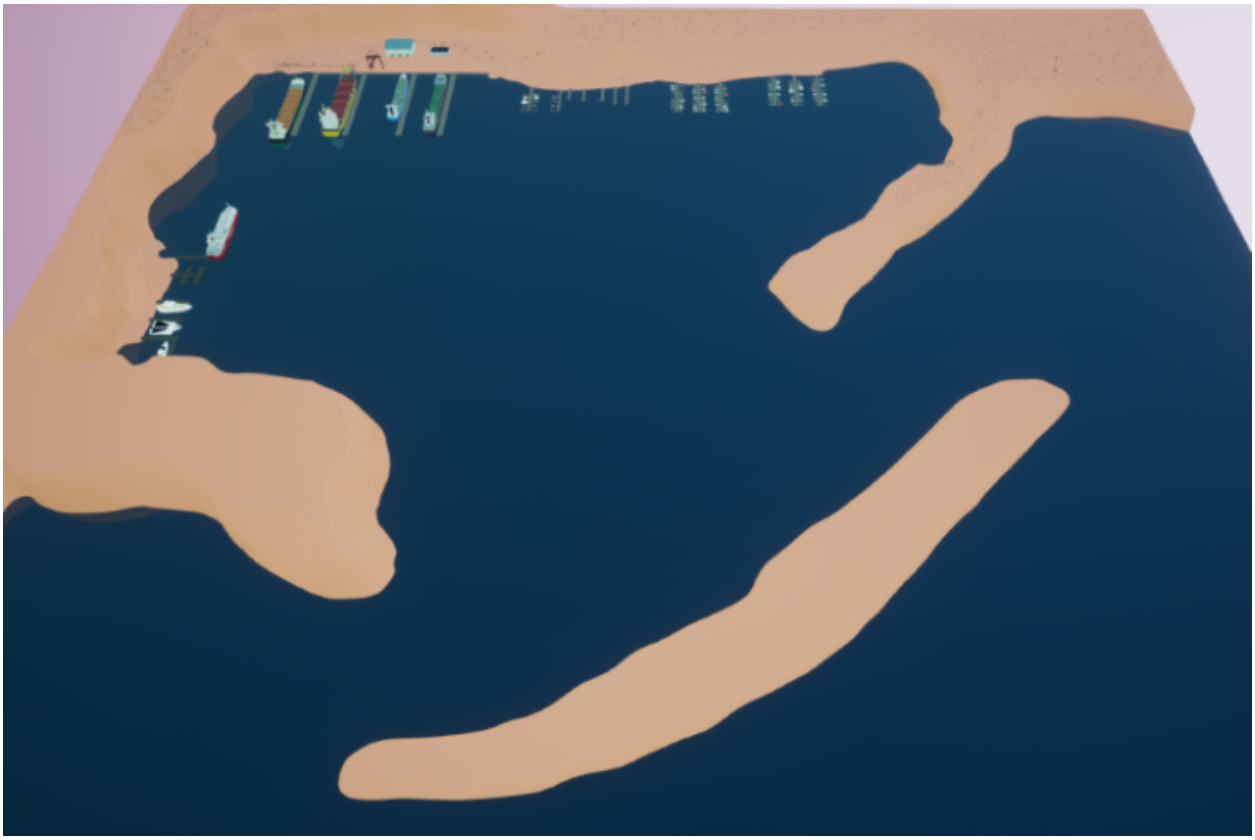
- **auv0: Main *TorpedoAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *SidescanSonar* configured with an azimuth of 170 degrees, 2000 range bins, range from 0.5 to 40 meters, and noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

OpenWater-TorpedoSinglebeamSonar

This scenario starts with a TorpedoAUV near a submarine. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *TorpedoAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *SinglebeamSonar* configured with an opening angle of 30 degrees, 200 range bins, noise, and a range from .5 to 30 meters.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

4.1.3 PierHarbor

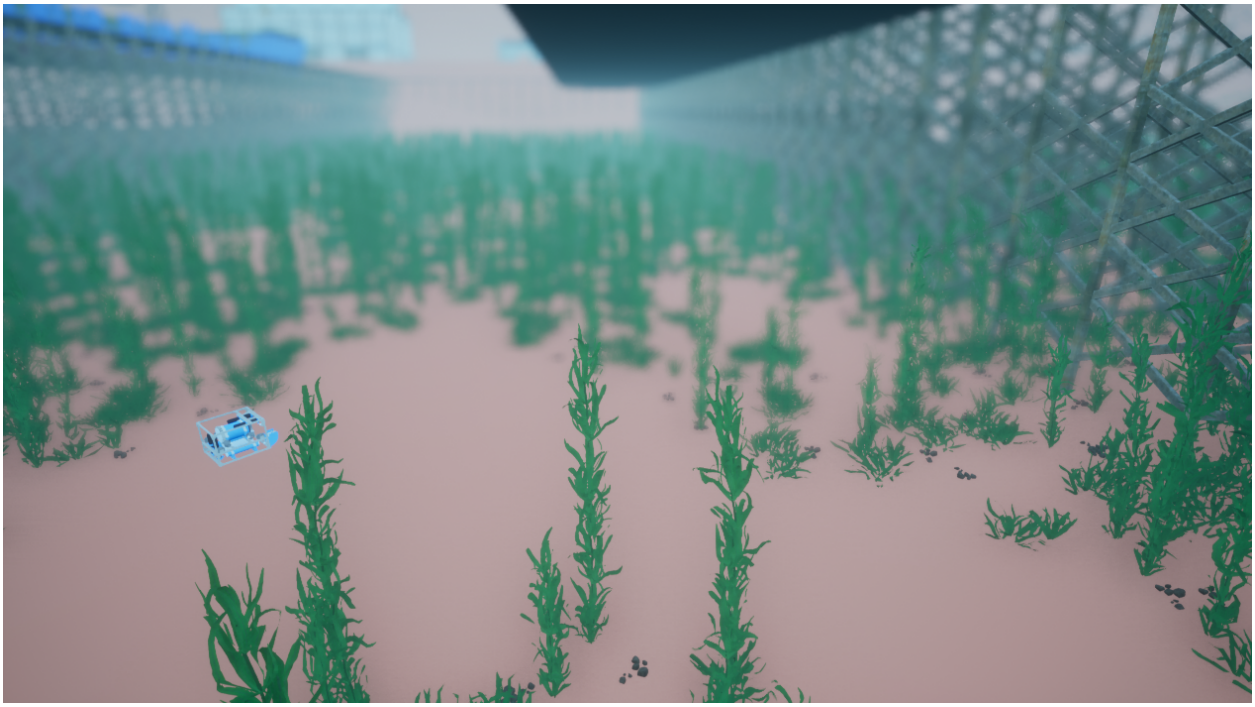


This environment has a number of different sized piers that can be inspected. Total environment size is about 2km x 2km. There is 3 sizes of piers ranging from small fishing docks to larger freight sizes, which boats included.

PierHarbor-Hovering

This scenario starts with a HoveringAUV near one of the larger docks. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

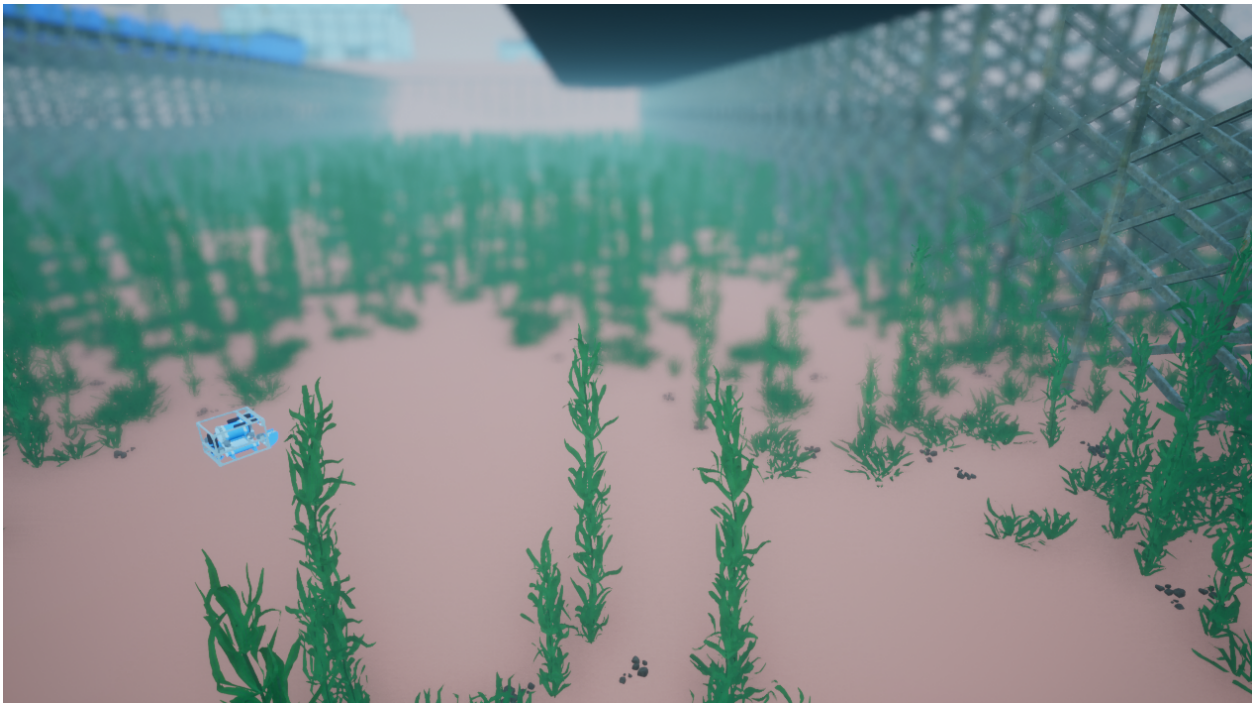
- **auv0: Main *HoveringAUV* agent**
  - `IMUSensor` configured with noise, bias, and returns bias.
  - `GPSSensor` gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - `DVLSensor` configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - `DepthSensor` configured with noise.
  - `PoseSensor` for ground truth.
  - `VelocitySensor` for ground truth.



PierHarbor-HoveringCamera

This scenario starts with a HoveringAUV near one of the larger docks. This is identical to the base version, only with cameras mounted. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

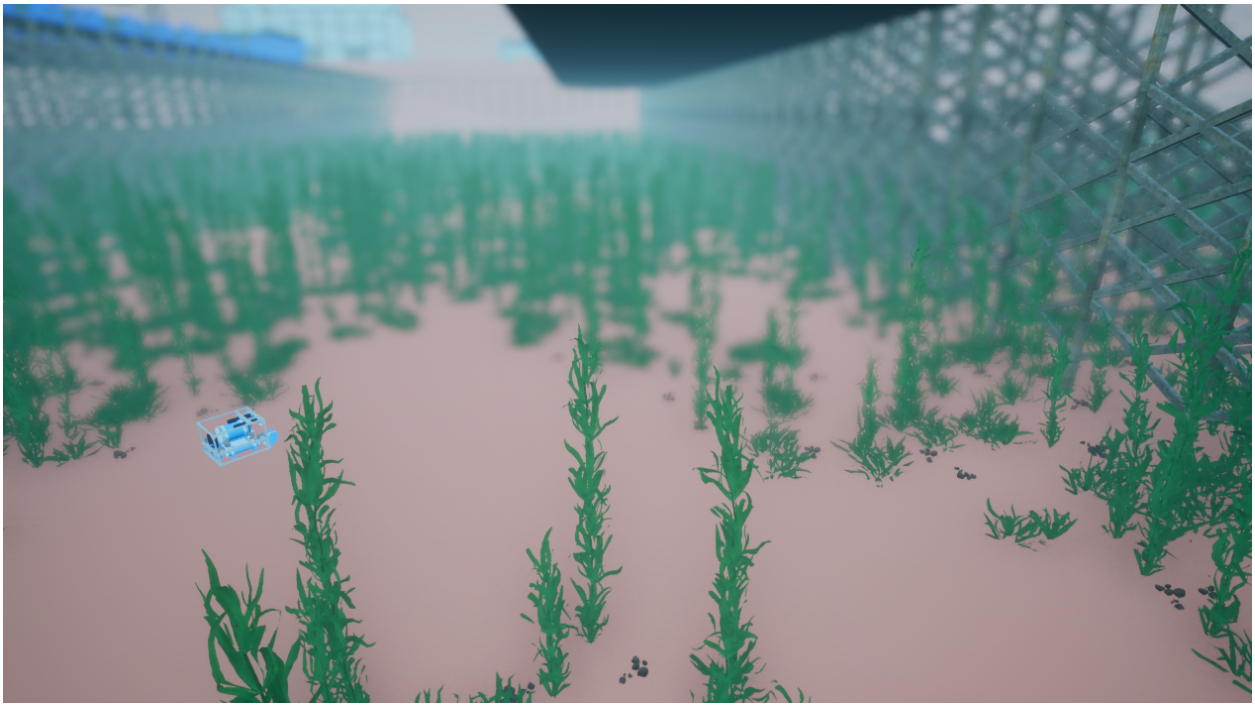
- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *RGBCamera* named LeftCamera.
  - *RGBCamera* named RightCamera.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.



PierHarbor-HoveringImagingSonar

This scenario starts with a HoveringAUV near one of the larger docks. This is identical to the base version, only with a sonar mounted. Octree leaf size is 2cm. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *ImagingSonar* configured with an elevation of 20 degrees, azimuth 130, range 1-40m, an initial octree generation of 40m, and all noise turned on.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

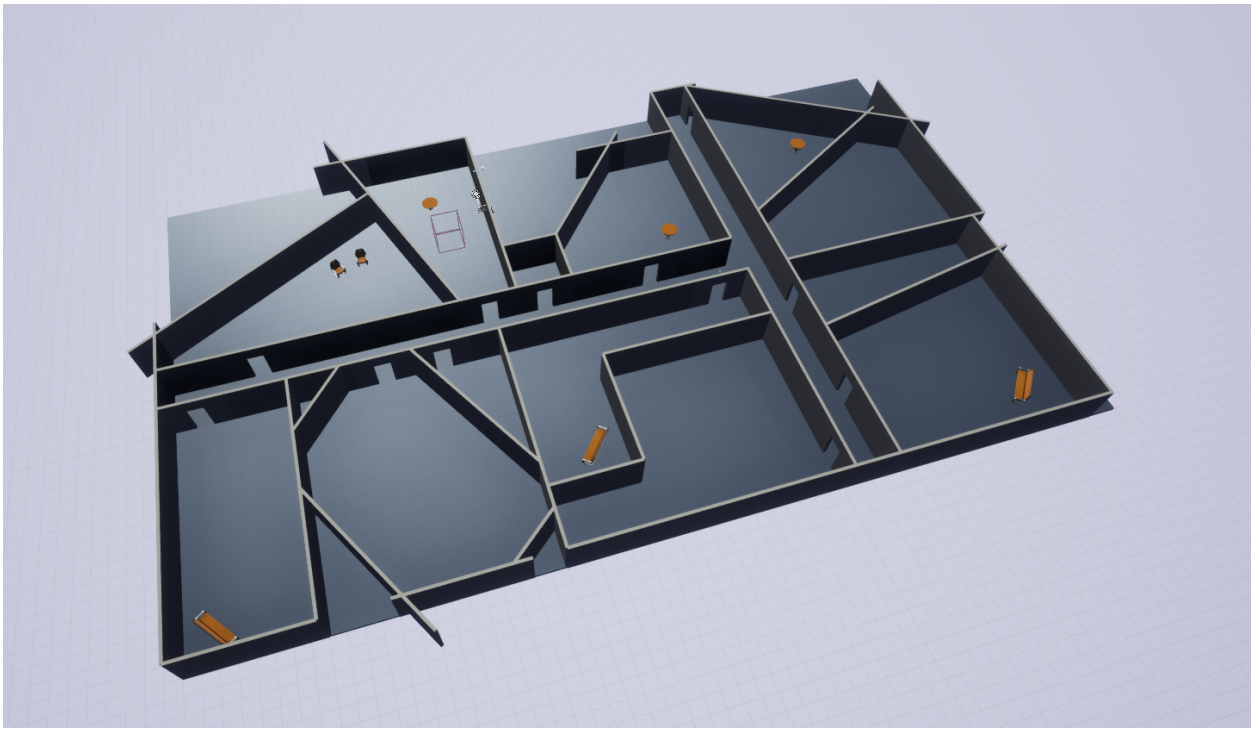


PierHarbor-Torpedo

This scenario starts with a HoveringAUV near one of the larger docks. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

4.1.4 Rooms



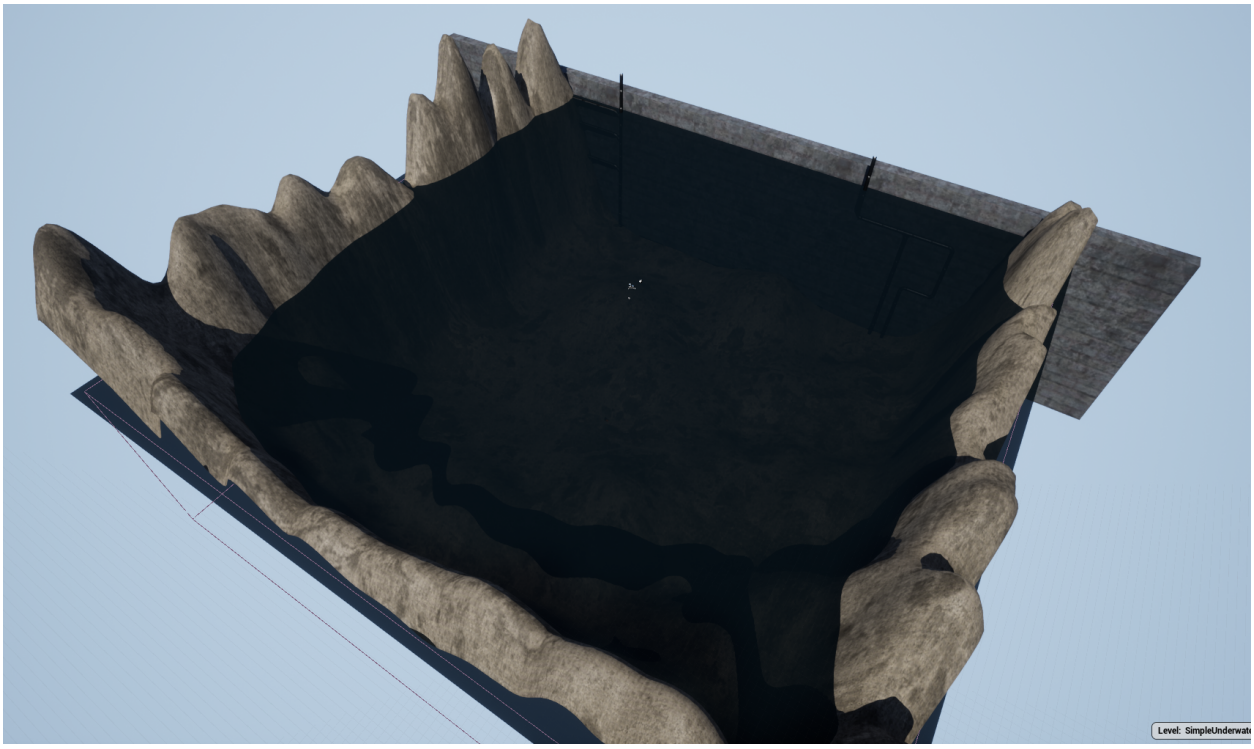
This is a simple world used for data generation in ECEN 522R, mobile robotics. It consists of various rooms for a TurtleAgent to navigate and generate data for development of various algorithms like occupancy grid mapping and particle filters.

Rooms-DataGen

This scenario starts with an TurtleAgent equipped with Location, Rotation and RangeFinder sensors. No noise is included in any of the sensors.

- **`turtle0`: Main *Turtle* agent**
  - *LocationSensor*
  - *RotationSensor*
  - `RangeFinder` configured with 64 beams and a max distance of 20m.

4.1.5 SimpleUnderwater



This is a basic underwater world to use for simulation purposes. It's equipped with pipes at one end to inspect, and very basic underwater imagery. It can be used when you need a lightweight environment and imagery doesn't matter.

SimpleUnderwater-Hovering

This scenario starts with a HoveringAUV in the center of the basin. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

SimpleUnderwater-Torpedo

This scenario starts with a TorpedoAUV in the center of the basin. Unless otherwise specified, all sensors are named the same as their class name, ie IMUSensor is named "IMUSensor".

- **auv0: Main *HoveringAUV* agent**
  - *IMUSensor* configured with noise, bias, and returns bias.
  - *GPSSensor* gets measurements with N(1, 0.25) of the surface, actual measurement also has noise.
  - *DVLSensor* configured with an elevation of 22.5 degrees, noise, and returns 4 range measurements.
  - *DepthSensor* configured with noise.
  - *PoseSensor* for ground truth.
  - *VelocitySensor* for ground truth.

4.2 Package Structure

A holoocean package is a `.zip` file containing a build of holoocean-engine that contains worlds and *Scenarios* for those worlds.

A package file is platform specific, since it contains a compiled binary of HoloOcean.

4.2.1 Package Contents

The `.zip` file must contain the following elements

1. A build of holoocean-engine
2. A `config.json` file that defines the worlds present in the package
3. Scenario configs for those worlds

4.2.2 Package Structure

The package.zip contains a `config.json` file at the root of the archive, as well as all of the scenarios for every world included in the package. The scenario files must follow the format `{WorldName}-{ScenarioName}.json`.

```
+package.zip
+-- config.json
+-- WorldName-ScenarioName.json
+-- LinuxNoEditor
    + UE4 build output
```

4.2.3 config.json

This configuration file contains the package-level configuration. Below is the format the config file is expected to follow:

`config.json`:

```
{
   "name": "{package_name}",
   "platform": "{Linux | Windows}",
   "version": "{package_version}",
   "path" : "{path to binary within the archive}",
   "worlds": [
      {
         "name": "{world_name}",
         "pre_start_steps": 2,
         "env_min": [-10, -10, -10],
         "env_max": [10, 10, 10]
      }
   ]
}
```

The `"pre_start_steps"` attribute for a world defines how many ticks should occur before starting the simulation, to work around world idiosyncrasies.

The `env_min`/`env_max` attributes are used to set the upper/lower bounds of the environment, used when an octree is made for a sonar sensor.

4.3 Package Installation Location

HoloOcean packages are by default saved in the current user profile, depending on the platform.

Platform	Location
Linux	`~/.local/share/holoocean/{holoocean_version}/worlds/`
Windows	`%USERPROFILE%\AppData\Local\holoocean\{holoocean_version}\worlds`

Note that the packages are saved in different subfolders based on the version of HoloOcean. This allows multiple versions of HoloOcean to coexist, without causing version incompatibility conflicts.

This is the path returned by *holoocean.util.get_holoocean_path()*

Each folder inside the worlds folder is considered a seperate package, so it must match the format of the archive described in *Package Contents*.

4.3.1 Overriding Location

The environment variable `HOLODECKPATH` can be set to override the default location given above.

> **Caution:** If `HOLODECKPATH` is used, it will override this version partitioning, so ensure that `HOLODECKPATH` only points to packages that are compatible with your version of Holodeck.

CHAPTER

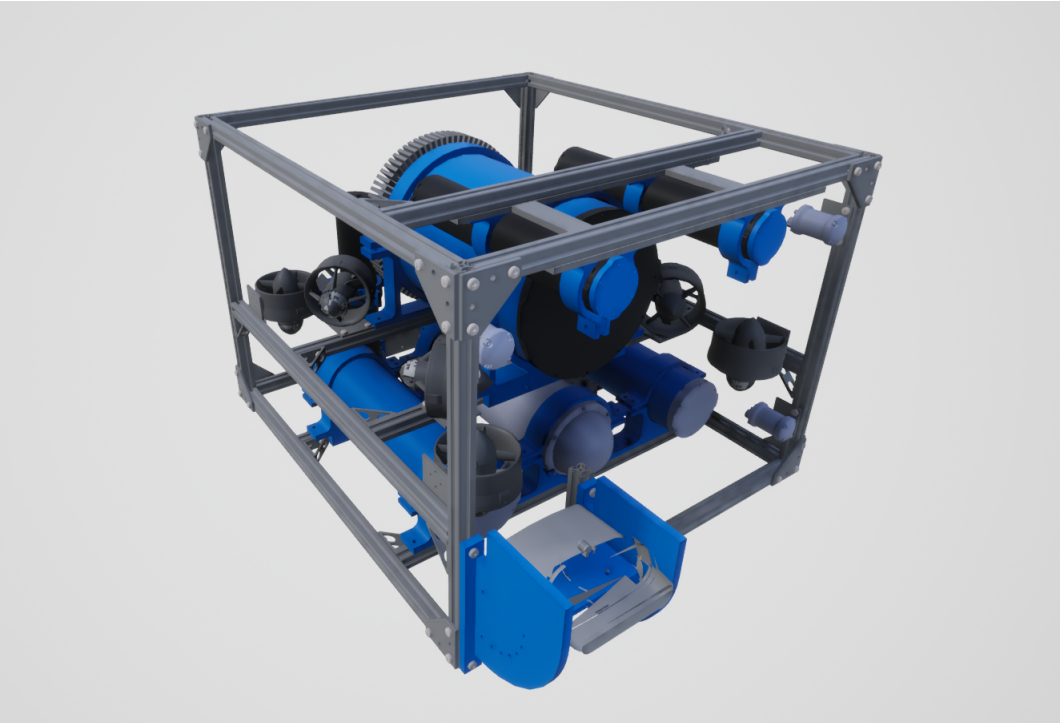
FIVE

HOLOOCEAN AGENTS

Documentation on specific agents available in HoloOcean:

5.1 HoveringAUV

5.1.1 Images



5.1.2 Description

Our custom in-house hovering AUV.

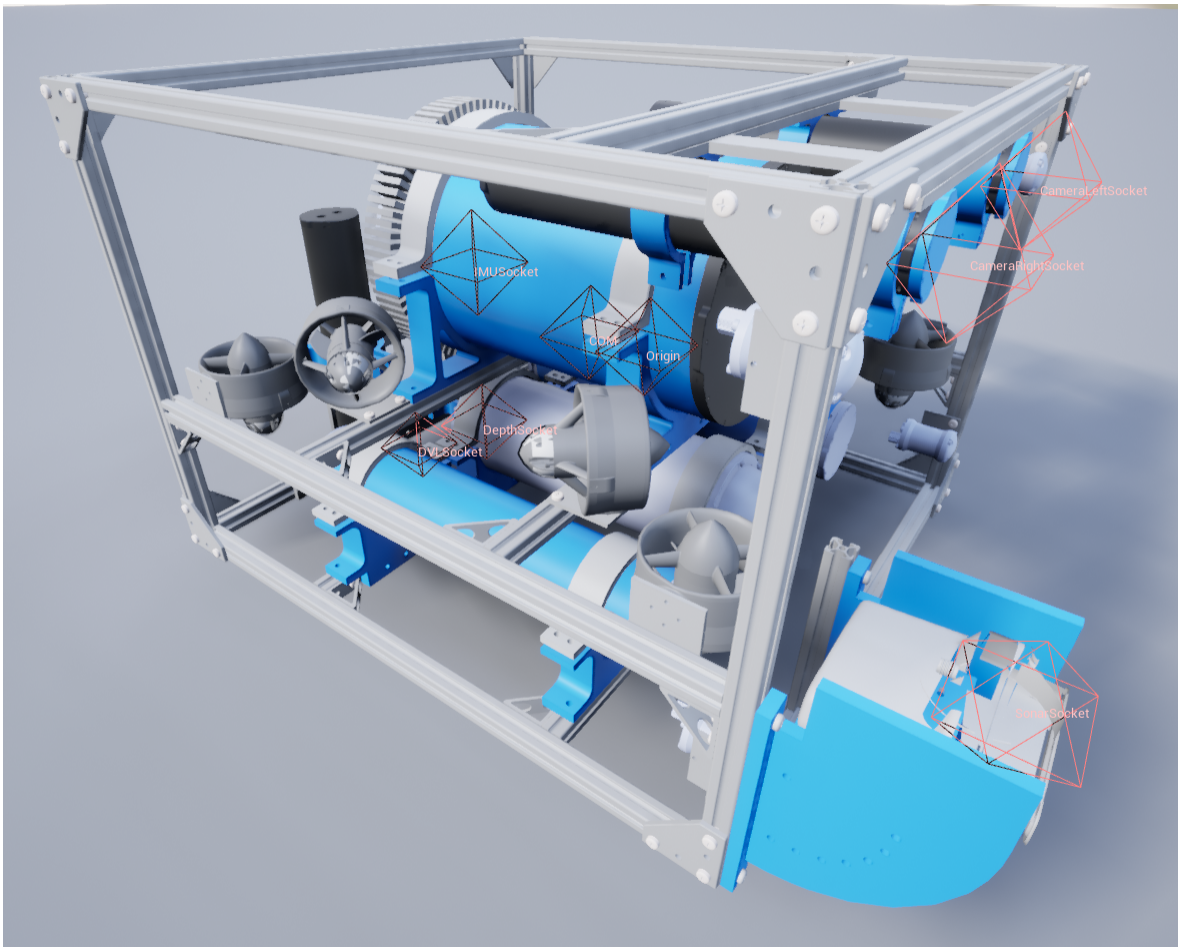
See the *HoveringAUV*.

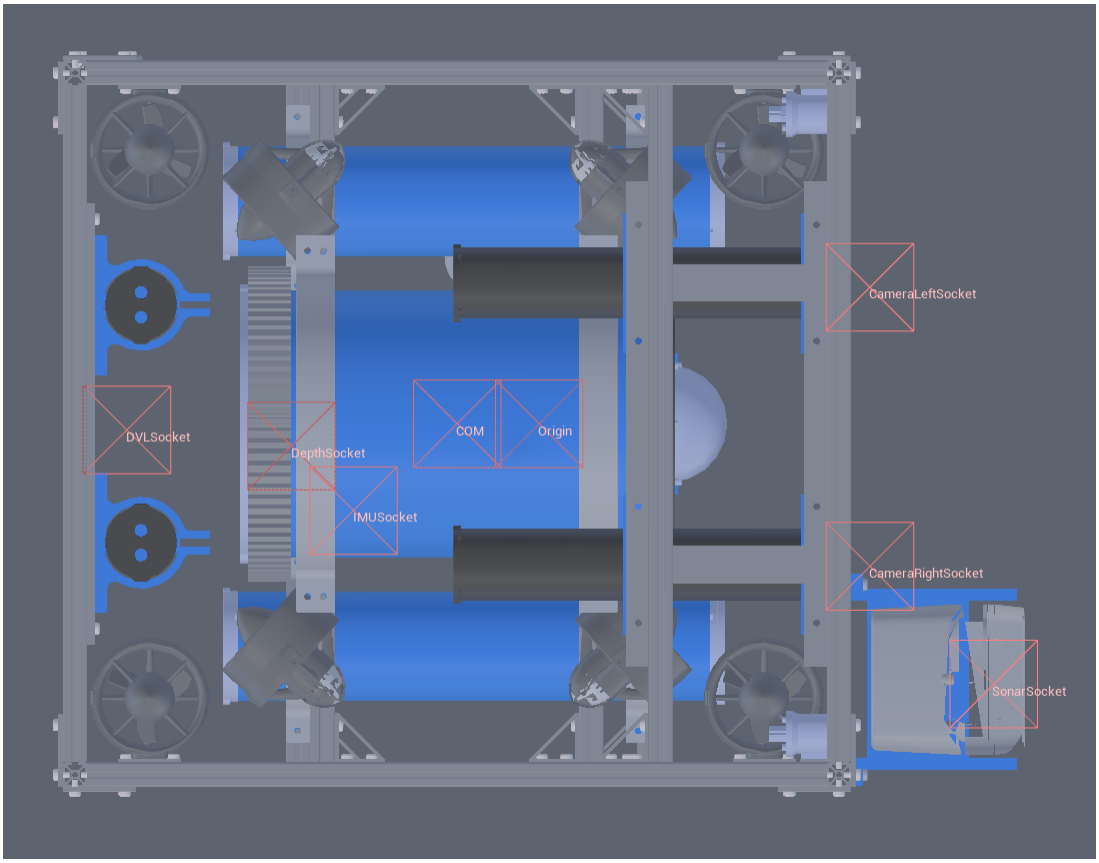
5.1.3 Control Schemes

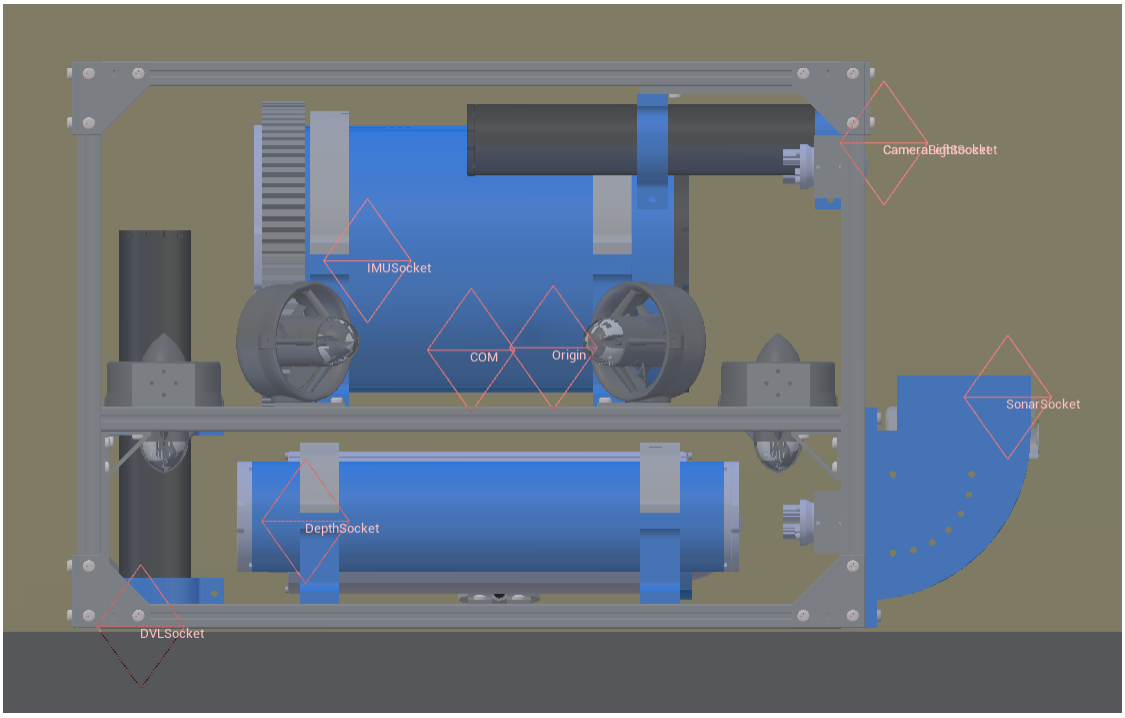
AUV Thrusters (``0``) An 8-length floating point vector used to specify the control on each thruster. They begin with the front right vertical thrusters, then goes around counter-clockwise, then repeat the last four with the sideways thrusters.

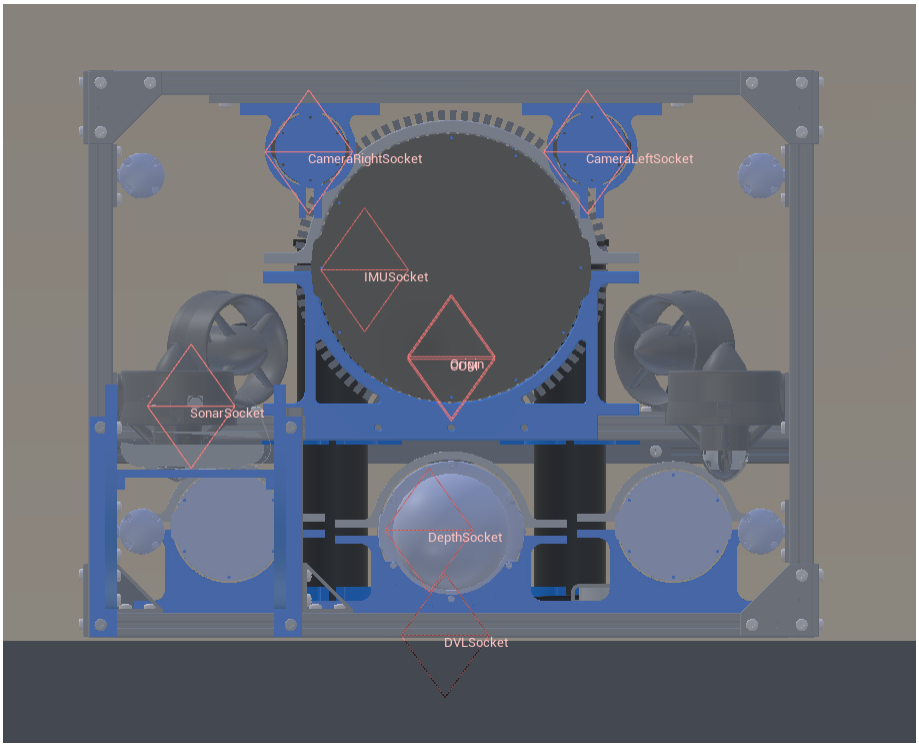
5.1.4 Sockets

- `COM` Center of mass
- `DVLSocket` Location of the DVL
- `IMUSocket` Location of the IMU.
- `DepthSocket` Location of the depth sensor.
- `SonarSocket` Location of the sonar sensor.
- `CameraRightSocket` Location of the left camera.
- `CameraLeftSocket` Location of the right camera.
- `Origin` true center of the robot
- `Viewport` where the robot is viewed from.


CameraLeftSocket
CameraRightSocket
IMUSocket
Origin
DepthSocket
DVLSocket
SonarSocket

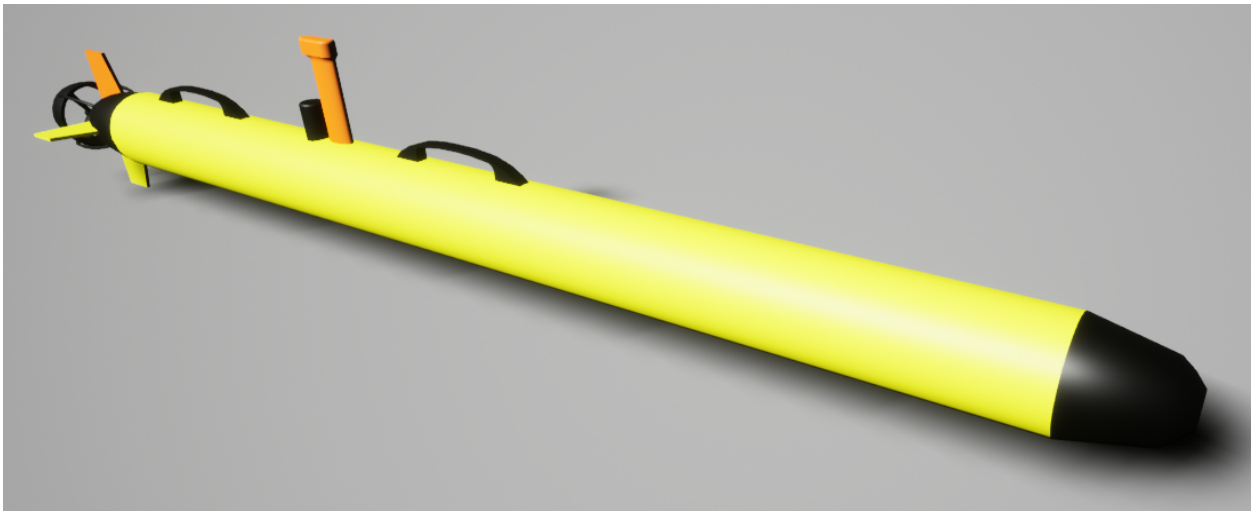

CameraLeftSocket
DVLSocket
DepthSocket
COM
Origin
IMUSocket
CameraRightSocket
SonarSocket


IMUSocket
COM
Origin
SonarSocket
DepthSocket
DVLSocket



5.2 TorpedoAUV

5.2.1 Images



5.2.2 Description

A generic AUV.

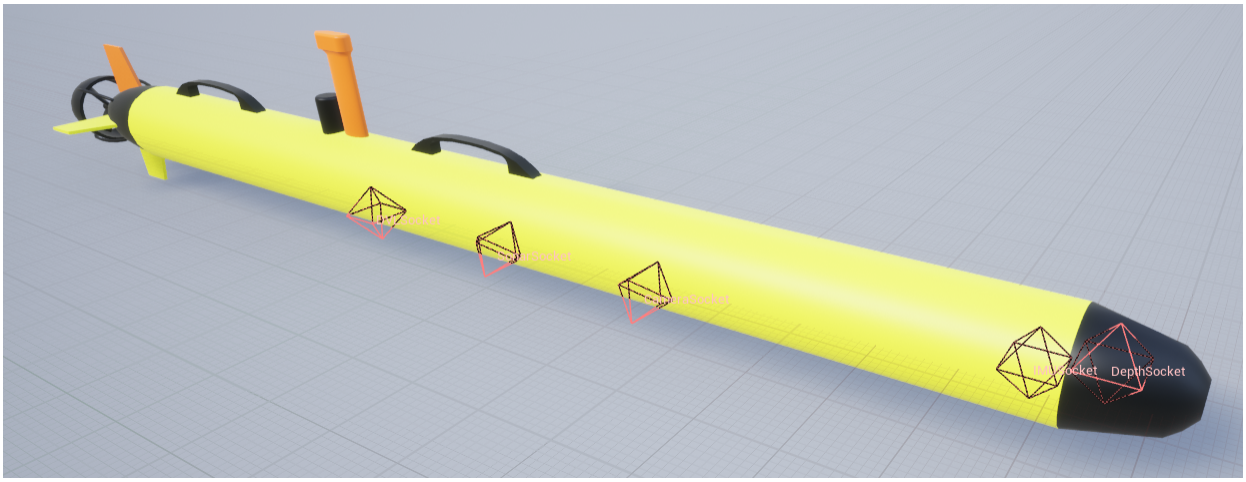
See the *TorpedoAUV*.

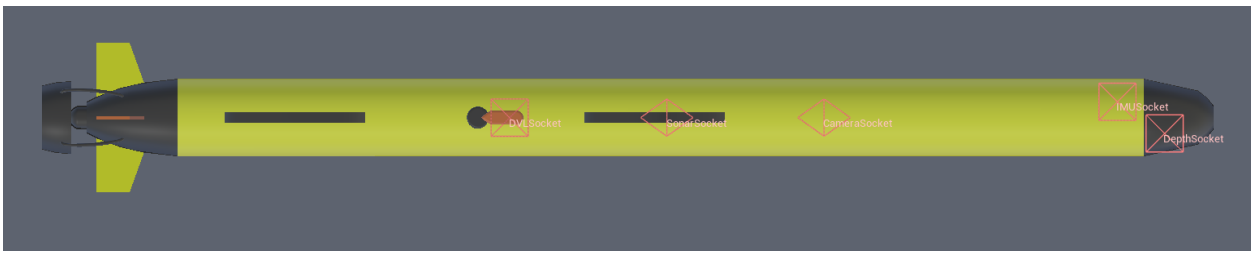
5.2.3 Control Schemes

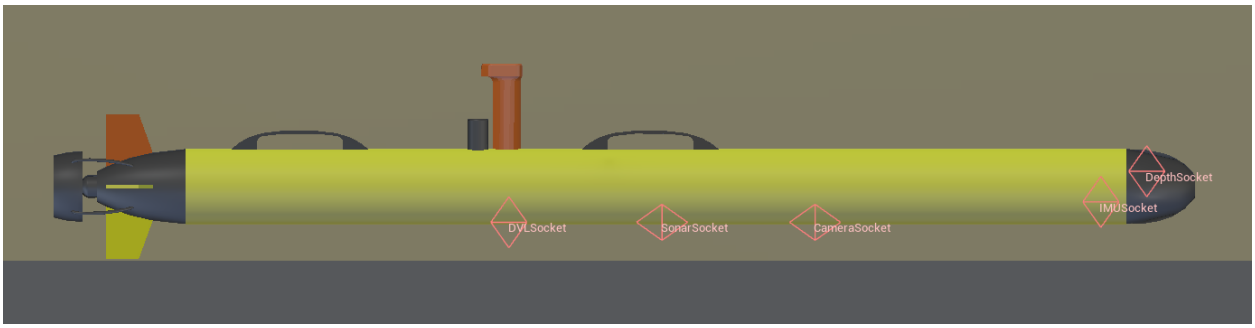
AUV Fins (``0``) Takes in a 5 length vector. The first element is the right fin angle from -45 to 45 degrees, then top, left, and bottom. The last element is the "thruster" with a value of -100 to 100.

5.2.4 Sockets

- `COM` Center of mass
- `DVLSocket` Location of the DVL
- `IMUSocket` Location of the IMU.
- `DepthSocket` Location of the depth sensor.
- `SonarSocket` Location of the sonar sensor.
- `Viewport` where the robot is viewed from.







5.3 TurtleAgent

5.3.1 Description

A simple turtle-bot agent with an arrow pointing forwards. Its radius is approximately 25cm and is approximately 10cm high.

The TurtleAgent moves when forces are applied to it - so it has momentum and mass, compared to the sphere-agent which teleports around. The TurtleAgent is subject to gravity and can climb ramps and slopes.

See *TurtleAgent* for more details.

5.3.2 Control Schemes

Sphere continuous (1) A 2-length floating point vector used to specify the agent's forward force (index 0) and rotation force (index 1).

5.3.3 Sockets

- `CameraSocket` located at the front of the body
- `Viewport` located behind the agent

5.4 UavAgent

5.4.1 Images

5.4.2 Description

A quadcopter UAV agent.

See the *UavAgent* class.

5.4.3 Control Schemes

UAV Torques (``0``) A 4-length floating point vector used to specify the pitch torque, roll torque, yaw torque and thrust with indices 0, 1, 2 and 3 respectively.

UAV Roll / Pitch / Yaw targets(``1``) A 4-length floating point vector used to specify the pitch, roll, yaw, and altitude targets. The values are specified in indices 0, 1, 2, and 3 respectively.

5.4.4 Sockets

- `CameraSocket` located underneath the uav body
- `Viewport` located behind the agent

CHAPTER

SIX

CHANGELOG

6.1 HoloOcean 0.5.0

3/22/2022

Large sonar upgrade!

6.1.1 Highlights

- 3 new sonar sensors
- Upgraded noise modeling in Imaging and Profiling Sonar
- Lots of misc bug fixes!
- Upgrade to Unreal Engine 4.27

6.1.2 New Features

- Added many new sonar sensors, including
  - *ImagingSonar*
  - *SidescanSonar*
  - *ProfilingSonar*
  - *SinglebeamSonar*
- ImagingSonar now has significantly improved noise modeling
- Can now specify a lifetime parameter to draw functions
- Now based on Unreal Engine 4.27

6.1.3 Changes

- SonarSensor is now ImagingSonar
- Rotations are now a correct [roll, pitch, yaw].
- step function no longer returns terminal, reward and info. Only state.
- Environments no longer publish over lcm in pre-start ticks.

6.1.4 Bug Fixes

- Timeouts are turned off for sonars to prevent premature termination of the simulation.
- Specifying a number of ticks to execute at once now ticks for all of them.

6.2 HoloOcean 0.4.1

9/21/2021

6.2.1 Bug Fixes

- Required pywin32 <= 228 for easier Windows install.

6.3 HoloOcean 0.4.0

9/17/2021

First official release!

6.3.1 Highlights

- New Ocean environment package.
- 2 new agents and 7 new sensors, along with updating of all previous sensors.
- Complete rebranding to HoloOcean.

6.3.2 New Features

- Added agents *HoveringAUV* and *TorpedoAUV*
- Added a plethora of new sensors, all with optional noise configurations
  - *ImagingSonar*
  - *DVLSensor*
  - *DepthSensor*
  - *GPSSensor*
  - *PoseSensor*

  - *AcousticBeaconSensor*
  - *OpticalModemSensor*
- New *Ocean* package.
- Added frame rate capping option.
- Added ticks_per_sec and frames_per_sec to scenario config, see *Frame Rates*.

6.3.3 Changes

- Everything is now rebranded from Holodeck -> HoloOcean.

6.3.4 Bug Fixes

- Sensors now return values from their location, not the agent location.
- IMU now returns angular velocity instead of linear velocity.
- Various integer -> float changes in scenario loading.

6.4 Pre-HoloOcean

See Holodeck changelog

CHAPTER

SEVEN

HOLOOCEAN

Module containing high level interface for loading environments.

Classes:

GL_VERSION()	OpenGL Version enum.

Functions:

make([scenario_name, scenario_cfg, ...])	Creates a HoloOcean environment

class holoocean.holoocean.**GL_VERSION**

OpenGL Version enum.

OPENGL3

The value for OpenGL3.

Type int

OPENGL4

The value for OpenGL4.

Type int

holoocean.holoocean.**make**(*scenario_name=''*, *scenario_cfg=None*, *gl_version=4*, *window_res=None*, *verbose=False*, *show_viewport=True*, *ticks_per_sec=None*, *frames_per_sec=None*, *copy_state=True*)

Creates a HoloOcean environment

Parameters

- **world_name** (str) – The name of the world to load as an environment. Must match the name of a world in an installed package.
- **scenario_cfg** (dict) – Dictionary containing scenario configuration, instead of loading a scenario from the installed packages. Dictionary should match the format of the JSON configuration files
- **gl_version** (int, optional) – The OpenGL version to use (Linux only). Defaults to GL_VERSION.OPENGL4.
- **window_res** ((int, int), optional) – The (height, width) to load the engine window at. Overrides the (optional) resolution in the scenario config file
- **verbose** (bool, optional) – Whether to run in verbose mode. Defaults to False.

- **show_viewport** (bool, optional) – If the viewport window should be shown on-screen (Linux only). Defaults to True
- **ticks_per_sec** (int, optional) – The number of frame ticks per unreal seconds. This will override whatever is in the configuration json. Defaults to 30.
- **frames_per_sec** (int or bool, optional) – The max number of frames ticks per real seconds. This will override whatever is in the configuration json. If True, will match ticks_per_sec. If False, will not be turned on. If an integer, will set to that value. Defaults to True.
- **copy_state** (bool, optional) – If the state should be copied or passed as a reference when returned. Defaults to True

Returns

A holoocean environment instantiated with all the settings necessary for the specified world, and other supplied arguments.

Return type *HoloOceanEnvironment*

CHAPTER

EIGHT

AGENTS

For a higher level description of the agents, see *HoloOcean Agents*.

Definitions for different agents that can be controlled from HoloOcean

Classes:

AgentDefinition(agent_name, agent_type[, ...])	Represents information needed to initialize agent.
AgentFactory()	Creates an agent object
ControlSchemes()	All allowed control schemes.
HoloOceanAgent(client[, name])	A learning agent in HoloOcean
HoveringAUV(client[, name])	A simple autonomous underwater vehicle.
TorpedoAUV(client[, name])	A simple foward motion autonomous underwater vehicle.
TurtleAgent(client[, name])	A simple turtle bot.
UavAgent(client[, name])	

class holoocean.agents.**AgentDefinition**(*agent_name*, *agent_type*, *sensors=None*, *starting_loc=(0, 0, 0)*, *starting_rot=(0, 0, 0)*, *existing=False*, *is_main_agent=False*)

Represents information needed to initialize agent.

Parameters

- **agent_name** (str) – The name of the agent to control.
- **agent_type** (str or type) – The type of HoloOceanAgent to control, string or class reference.
- **sensors** (*SensorDefinition* or class type (if no duplicate sensors)) – A list of HoloOceanSensors to read from this agent.
- **starting_loc** (list of float) – Starting [x, y, z] location for agent (see *Coordinate System*)
- **starting_rot** (list of float) – Starting [roll, pitch, yaw] rotation for agent (see *Rotations*)
- **existing** (bool) – If the agent exists in the world or not (deprecated)

class holoocean.agents.**AgentFactory**

Creates an agent object

Methods:

build_agent(client, agent_def)	Constructs an agent

static build_agent(*client*, *agent_def*)
Constructs an agent

Parameters

- **client** (*holoocean.holooceanclient.HoloOceanClient*) – HoloOceanClient agent is associated with
- **agent_def** (*AgentDefinition*) – Definition of the agent to instantiate

Returns:

class holoocean.agents.**ControlSchemes**
All allowed control schemes.

ANDROID_TORQUES
Default Android control scheme. Specify a torque for each joint.

Type int

CONTINUOUS_SPHERE_DEFAULT
Default ContinuousSphere control scheme. Takes two commands, [forward_delta, turn_delta].

Type int

DISCRETE_SPHERE_DEFAULT
Default DiscreteSphere control scheme. Takes a value, 0-4, which corresponds with forward, backward, right, and left.

Type int

NAV_TARGET_LOCATION
Default NavAgent control scheme. Takes a target xyz coordinate.

Type int

UAV_TORQUES
Default UAV control scheme. Takes torques for roll, pitch, and yaw, as well as thrust.

Type int

UAV_ROLL_PITCH_YAW_RATE_ALT
Control scheme for UAV. Takes roll, pitch, yaw rate, and altitude targets.

Type int

HAND_AGENT_MAX_TORQUES
Default Android control scheme. Specify a torque for each joint.

Type int

class holoocean.agents.**HoloOceanAgent**(*client*, *name='DefaultAgent'*)
A learning agent in HoloOcean

Agents can act, receive rewards, and receive observations from their sensors. Examples include the Android, UAV, and SphereRobot.

Parameters

- **client** (*HoloOceanClient*) – The HoloOceanClient that this agent belongs with.
- **name** (str, optional) – The name of the agent. Must be unique from other agents in the same environment.

- **sensors** (dict of (str, *HoloOceanSensor*)) – A list of HoloOceanSensors to read from this agent.

name
The name of the agent.

Type str

sensors
List of HoloOceanSensors on this agent.

Type dict of (string, *HoloOceanSensor*)

agent_state_dict
A dictionary that maps sensor names to sensor observation data.

Type dict

Methods:

act(action)	Sets the command for the agent.
add_sensors(sensor_defs)	Adds a sensor to a particular agent object and attaches an instance of the sensor to the agent in the world.
clear_action()	Sets the action to zeros, effectively removing any previous actions.
get_joint_constraints(joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the specified joint.
has_camera()	Indicatates whether this agent has a camera or not.
remove_sensors(sensor_defs)	Removes a sensor from a particular agent object and detaches it from the agent in the world.
set_control_scheme(index)	Sets the control scheme for the agent.
set_physics_state(location, rotation, ...)	Sets the location, rotation, velocity and angular velocity of an agent.
teleport([location, rotation])	Teleports the agent to a specific location, with a specific rotation.

Attributes:

action_space	Gets the action space for the current agent and control scheme.
control_schemes	A list of all control schemes for the agent.

act(*action*)
Sets the command for the agent. Action depends on the agent type and current control scheme.

Parameters **action** (np.ndarray) – The action to take.

property action_space
Gets the action space for the current agent and control scheme.

Returns

The action space for this agent and control scheme.

Return type *ActionSpace*

add_sensors(*sensor_defs*)
Adds a sensor to a particular agent object and attaches an instance of the sensor to the agent in the world.

:param sensor_defs (*HoloOceanSensor* or: list of *HoloOceanSensor*): Sensors to add to the agent.

`clear_action()`
Sets the action to zeros, effectively removing any previous actions.

`property control_schemes`
A list of all control schemes for the agent. Each list element is a 2-tuple, with the first element containing a short description of the control scheme, and the second element containing the `ActionSpace` for the control scheme.

Returns Each tuple contains a short description and the ActionSpace

Return type (`str`, *ActionSpace*)

`get_joint_constraints(joint_name)`
Returns the corresponding swing1, swing2 and twist limit values for the specified joint. Will return None if the joint does not exist for the agent.

Returns obj)

Return type

(

`has_camera()`
Indicatates whether this agent has a camera or not.

Returns If the agent has a sensor or not

Return type `bool`

`remove_sensors(sensor_defs)`
Removes a sensor from a particular agent object and detaches it from the agent in the world.

:param sensor_defs (*HoloOceanSensor* or: list of *HoloOceanSensor*): Sensors to remove from the agent.

`set_control_scheme(index)`
Sets the control scheme for the agent. See *ControlSchemes*.

Parameters **`index`** (`int`) – The control scheme to use. Should be set with an enum from *ControlSchemes*.

`set_physics_state(location, rotation, velocity, angular_velocity)`
Sets the location, rotation, velocity and angular velocity of an agent.

Parameters

- **`location`** (*np.ndarray*) – New location (`[x, y, z]` (see *Coordinate System*))
- **`rotation`** (*np.ndarray*) – New rotation (`[roll, pitch, yaw]`, see (see *Rotations*))
- **`velocity`** (*np.ndarray*) – New velocity (`[x, y, z]` (see *Coordinate System*))
- **`angular_velocity`** (*np.ndarray*) – New angular velocity (`[x, y, z]` in **degrees** (see *Coordinate System*))

`teleport(location=None, rotation=None)`
Teleports the agent to a specific location, with a specific rotation.

Parameters

- **`location`** (*np.ndarray, optional*) – An array with three elements specifying the target world coordinates `[x, y, z]` in meters (see *Coordinate System*).

  If `None` (default), keeps the current location.

- **rotation** (*np.ndarray, optional*) – An array with three elements specifying roll, pitch, and yaw in degrees of the agent.

  If None (default), keeps the current rotation.

class holoocean.agents.**HoveringAUV**(*client*, *name='DefaultAgent'*)

A simple autonomous underwater vehicle.

Action Space::

```
[Vertical Front Starboard, Vertical Front Port, Vertical Back Port, Vertical Back␣
↪Starboard,
Angled Front Starboard, Angled Front Port, Angled Back Port, Angled Back Starboard]
```

- All are capped by max acceleration

Inherits from *HoloOceanAgent*.

Attributes:

control_schemes	A list of all control schemes for the agent.

Methods:

get_joint_constraints(joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the specified joint.

property control_schemes

A list of all control schemes for the agent. Each list element is a 2-tuple, with the first element containing a short description of the control scheme, and the second element containing the ActionSpace for the control scheme.

Returns Each tuple contains a short description and the ActionSpace

Return type (str, *ActionSpace*)

get_joint_constraints(*joint_name*)

Returns the corresponding swing1, swing2 and twist limit values for the specified joint. Will return None if the joint does not exist for the agent.

Returns obj)

Return type

(

class holoocean.agents.**TorpedoAUV**(*client*, *name='DefaultAgent'*)

A simple foward motion autonomous underwater vehicle.

Action Space::

```
[left_fin, top_fin, right_fin, bottom_fin, thrust]
```

- All are capped by max acceleration

Inherits from *HoloOceanAgent*.

Attributes:

control_schemes	A list of all control schemes for the agent.

Methods:

get_joint_constraints(joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the specified joint.

property control_schemes
A list of all control schemes for the agent. Each list element is a 2-tuple, with the first element containing a short description of the control scheme, and the second element containing the `ActionSpace` for the control scheme.

Returns Each tuple contains a short description and the ActionSpace

Return type (`str`, *ActionSpace*)

get_joint_constraints(*joint_name*)
Returns the corresponding swing1, swing2 and twist limit values for the specified joint. Will return None if the joint does not exist for the agent.

Returns obj)

Return type

(

class holoocean.agents.**TurtleAgent**(*client*, *name='DefaultAgent'*)
A simple turtle bot.

Action Space:

`[forward_force, rot_force]`

- `forward_force` is capped at 160 in either direction
- `rot_force` is capped at 35 either direction

Inherits from *HoloOceanAgent*.

Attributes:

control_schemes	A list of all control schemes for the agent.

Methods:

get_joint_constraints(joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the specified joint.

property control_schemes
A list of all control schemes for the agent. Each list element is a 2-tuple, with the first element containing a short description of the control scheme, and the second element containing the `ActionSpace` for the control scheme.

Returns Each tuple contains a short description and the ActionSpace

Return type (`str`, *ActionSpace*)

get_joint_constraints(*joint_name*)
Returns the corresponding swing1, swing2 and twist limit values for the specified joint. Will return None

if the joint does not exist for the agent.

Returns obj)

Return type

(

class holoocean.agents.**UavAgent**(*client*, *name='DefaultAgent'*)

Attributes:

control_schemes	A list of all control schemes for the agent.

Methods:

get_joint_constraints(joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the specified joint.

property control_schemes

A list of all control schemes for the agent. Each list element is a 2-tuple, with the first element containing a short description of the control scheme, and the second element containing the `ActionSpace` for the control scheme.

Returns Each tuple contains a short description and the ActionSpace

Return type (`str`, *ActionSpace*)

get_joint_constraints(*joint_name*)

Returns the corresponding swing1, swing2 and twist limit values for the specified joint. Will return None if the joint does not exist for the agent.

Returns obj)

Return type

(

CHAPTER

NINE

ENVIRONMENTS

Module containing the environment interface for HoloOcean. An environment contains all elements required to communicate with a world binary or HoloOceanCore editor.

It specifies an environment, which contains a number of agents, and the interface for communicating with the agents.

Classes:

HoloOceanEnvironment([agent_definitions, ...])	Proxy for communicating with a HoloOcean world

class holoocean.environments.**HoloOceanEnvironment**(*agent_definitions=None*, *binary_path=None*, *window_size=None*, *start_world=True*, *uuid=''*, *gl_version=4*, *verbose=False*, *pre_start_steps=2*, *show_viewport=True*, *ticks_per_sec=None*, *frames_per_sec=None*, *copy_state=True*, *scenario=None*)

Proxy for communicating with a HoloOcean world

Instantiate this object using *holoocean.holoocean.make()*.

Parameters

- **agent_definitions** (list of AgentDefinition) – Which agents are already in the environment
- **binary_path** (str, optional) – The path to the binary to load the world from. Defaults to None.
- **window_size** ((int,:obj:*int*)) – height, width of the window to open
- **start_world** (bool, optional) – Whether to load a binary or not. Defaults to True.
- **uuid** (str) – A unique identifier, used when running multiple instances of holoocean. Defaults to "".
- **gl_version** (int, optional) – The version of OpenGL to use for Linux. Defaults to 4.
- **verbose** (bool) – If engine log output should be printed to stdout
- **pre_start_steps** (int) – Number of ticks to call after initializing the world, allows the level to load and settle.
- **show_viewport** (bool, optional) – If the viewport should be shown (Linux only) Defaults to True.
- **ticks_per_sec** (int, optional) – The number of frame ticks per unreal seconds. This will override whatever is in the configuration json. Defaults to 30.

- **frames_per_sec** (int or bool, optional) – The max number of frames ticks per real seconds. This will override whatever is in the configuration json. If True, will match ticks_per_sec. If False, will not be turned on. If an integer, will set to that value. Defaults to true.
- **copy_state** (bool, optional) – If the state should be copied or returned as a reference. Defaults to True.
- **scenario** (dict) – The scenario that is to be loaded. See *Scenario File Format* for the schema.

Methods:

act(agent_name, action)	Supplies an action to a particular agent, but doesn't tick the environment.
add_agent(agent_def[, is_main_agent])	Add an agent in the world.
draw_arrow(start, end[, color, thickness, ...])	Draws a debug arrow in the world
draw_box(center, extent[, color, thickness, ...])	Draws a debug box in the world
draw_line(start, end[, color, thickness, ...])	Draws a debug line in the world
draw_point(loc[, color, thickness, lifetime])	Draws a debug point in the world
get_joint_constraints(agent_name, joint_name)	Returns the corresponding swing1, swing2 and twist limit values for the
get_reward_terminal()	Returns the reward and terminal state
info()	Returns a string with specific information about the environment.
move_viewport(location, rotation)	Teleport the camera to the given location
reset()	Resets the environment, and returns the state.
send_acoustic_message(id_from, id_to, ...)	Send a message from one beacon to another.
send_optical_message(id_from, id_to, msg_data)	Sends data between various instances of OpticalModemSensor
send_world_command(name[, num_params, ...])	Send a world command.
set_control_scheme(agent_name, control_scheme)	Set the control scheme for a specific agent.
set_render_quality(render_quality)	Adjusts the rendering quality of HoloOcean.
should_render_viewport(render_viewport)	Controls whether the viewport is rendered or not
spawn_prop(prop_type[, location, rotation, ...])	Spawns a basic prop object in the world like a box or sphere.
step(action[, ticks, publish])	Supplies an action to the main agent and tells the environment to tick once.
tick([num_ticks, publish])	Ticks the environment once.

Attributes:

action_space	Gives the action space for the main agent.
beacons	Gets all instances of AcousticBeaconSensor in the environment.
beacons_id	Gets all ids of AcousticBeaconSensor in the environment.
beacons_status	Gets all status of AcousticBeaconSensor in the environment.
modems	Gets all instances of OpticalModemSensor in the environment.

continues on next page

Table 3 – continued from previous page

modems_id	Gets all ids of OpticalModemSensor in the environment.

act(*agent_name*, *action*)

Supplies an action to a particular agent, but doesn't tick the environment. Primary mode of interaction for multi-agent environments. After all agent commands are supplied, they can be applied with a call to *tick*.

Parameters

- **agent_name** (`str`) – The name of the agent to supply an action for.
- **action** (`np.ndarray` or `list`) – The action to apply to the agent. This action will be applied every time *tick* is called, until a new action is supplied with another call to act.

property action_space

Gives the action space for the main agent.

Returns The action space for the main agent.

Return type *ActionSpace*

add_agent(*agent_def*, *is_main_agent=False*)

Add an agent in the world.

It will be spawn when *tick()* or *step()* is called next.

The agent won't be able to be used until the next frame.

Parameters

- **agent_def** (*AgentDefinition*) – The definition of the agent to
- **spawn.** –

property beacons

Gets all instances of AcousticBeaconSensor in the environment.

Returns List of all AcousticBeaconSensor in environment

Return type (`list` of `AcousticBeaconSensor`)

property beacons_id

Gets all ids of AcousticBeaconSensor in the environment.

Returns List of all AcousticBeaconSensor ids in environment

Return type (`list` of `int`)

property beacons_status

Gets all status of AcousticBeaconSensor in the environment.

Returns List of all AcousticBeaconSensor status in environment

Return type (`list` of `str`)

draw_arrow(*start*, *end*, *color=None*, *thickness=10.0*, *lifetime=1.0*)

Draws a debug arrow in the world

Parameters

- **start** (`list` of `float`) – The start `[x, y, z]` location in meters of the line. (see *Coordinate System*)

- **end** (list of float) – The end [x, y, z] location in meters of the arrow
- **color** (list) – [r, g, b] color value (from 0 to 255). Defaults to [255, 0, 0] (red).
- **thickness** (float) – Thickness of the arrow. Defaults to 10.
- **lifetime** (float) – Number of simulation seconds the object should persist. If 0, makes persistent. Defaults to 1.

draw_box(*center*, *extent*, *color=None*, *thickness=10.0*, *lifetime=1.0*)
Draws a debug box in the world

Parameters

- **center** (list of float) – The start [x, y, z] location in meters of the box. (see *Coordinate System*)
- **extent** (list of float) – The [x, y, z] extent of the box
- **color** (list) – [r, g, b] color value (from 0 to 255). Defaults to [255, 0, 0] (red).
- **thickness** (float) – Thickness of the lines. Defaults to 10.
- **lifetime** (float) – Number of simulation seconds the object should persist. If 0, makes persistent. Defaults to 1.

draw_line(*start*, *end*, *color=None*, *thickness=10.0*, *lifetime=1.0*)
Draws a debug line in the world

Parameters

- **start** (list of float) – The start [x, y, z] location in meters of the line. (see *Coordinate System*)
- **end** (list of float) – The end [x, y, z] location in meters of the line
- **color** (list`) – [r, g, b] color value (from 0 to 255). Defaults to [255, 0, 0] (red).
- **thickness** (float) – Thickness of the line. Defaults to 10.
- **lifetime** (float) – Number of simulation seconds the object should persist. If 0, makes persistent. Defaults to 1.

draw_point(*loc*, *color=None*, *thickness=10.0*, *lifetime=1.0*)
Draws a debug point in the world

Parameters

- **loc** (list of float) – The [x, y, z] start of the box. (see *Coordinate System*)
- **color** (list of float) – [r, g, b] color value (from 0 to 255). Defaults to [255, 0, 0] (red).
- **thickness** (float) – Thickness of the point. Defaults to 10.
- **lifetime** (float) – Number of simulation seconds the object should persist. If 0, makes persistent. Defaults to 1.

get_joint_constraints(*agent_name*, *joint_name*)

Returns the corresponding swing1, swing2 and twist limit values for the specified agent and joint. Will return None if the joint does not exist for the agent.

Returns np.ndarray

get_reward_terminal()
Returns the reward and terminal state

Returns

A 2tuple:

- Reward (`float`): Reward returned by the environment.
- Terminal: The bool terminal signal returned by the environment.

Return type (`float`, `bool`)

info()
Returns a string with specific information about the environment. This information includes which agents are in the environment and which sensors they have.

Returns Information in a string format.

Return type `str`

property modems
Gets all instances of OpticalModemSensor in the environment.

Returns List of all OpticalModemSensor in environment

Return type (`list` of `OpticalModemSensor`)

property modems_id
Gets all ids of OpticalModemSensor in the environment.

Returns List of all OpticalModemSensor ids in environment

Return type (`list` of `int`)

move_viewport(*location*, *rotation*)
Teleport the camera to the given location

By the next tick, the camera's location and rotation will be updated

Parameters

- **location** (`list` of `float`) – The `[x, y, z]` location to give the camera (see *Coordinate System*)
- **rotation** (`list` of `float`) – The x-axis that the camera should look down. Other 2 axes are formed by a horizontal y-aixs, and then the corresponding z-axis. (see *Rotations*)

reset()
Resets the environment, and returns the state. If it is a single agent environment, it returns that state for that agent. Otherwise, it returns a dict from agent name to state.

Returns Returns the same as *tick*.

Return type `tuple` or `dict`

send_acoustic_message(*id_from*, *id_to*, *msg_type*, *msg_data*)
Send a message from one beacon to another.

Parameters

- **id_from** (`int`) – The integer ID of the transmitting modem.
- **id_to** (`int`) – The integer ID of the receiving modem.
- **msg_type** (`str`) – The message type. See *holoocean.sensors.AcousticBeaconSensor* for a list.

- **msg_data** – The message to be transmitted. Currently can be any python object.

send_optical_message(*id_from*, *id_to*, *msg_data*)
Sends data between various instances of OpticalModemSensor

Parameters

- **id_from** (int) – The integer ID of the transmitting modem.
- **id_to** (int) – The integer ID of the receiving modem.
- **msg_data** – The message to be transmitted. Currently can be any python object.

send_world_command(*name*, *num_params=None*, *string_params=None*)
Send a world command.

A world command sends an abitrary command that may only exist in a specific world or package. It is given a name and any amount of string and number parameters that allow it to alter the state of the world.

If a command is sent that does not exist in the world, the environment will exit.

Parameters

- **name** (str) – The name of the command, ex "OpenDoor"
- **(obj** (*string_params*) – *list* of int): List of arbitrary number parameters
- **(obj** – *list* of string): List of arbitrary string parameters

set_control_scheme(*agent_name*, *control_scheme*)
Set the control scheme for a specific agent.

Parameters

- **agent_name** (str) – The name of the agent to set the control scheme for.
- **control_scheme** (int) – A control scheme value (see *ControlSchemes*)

set_render_quality(*render_quality*)
Adjusts the rendering quality of HoloOcean.

Parameters **render_quality** (int) – An integer between 0 = Low Quality and 3 = Epic quality.

should_render_viewport(*render_viewport*)
Controls whether the viewport is rendered or not

Parameters **render_viewport** (boolean) – If the viewport should be rendered

spawn_prop(*prop_type*, *location=None*, *rotation=None*, *scale=1*, *sim_physics=False*, *material=''*, *tag=''*)
Spawns a basic prop object in the world like a box or sphere.

Prop will not persist after environment reset.

Parameters

- **prop_type** (string) – The type of prop to spawn. Can be box, sphere, cylinder, or cone.
- **location** (list of float) – The [x, y, z] location of the prop.
- **rotation** (list of float) – The [roll, pitch, yaw] rotation of the prop.
- **scale** (list of float) or (float) – The [x, y, z] scalars to the prop size, where the default size is 1 meter. If given a single float value, then every dimension will be scaled to that value.
- **sim_physics** (boolean) – Whether the object is mobile and is affected by gravity.

- **material** (string) – The type of material (texture) to apply to the prop. Can be white, gold, cobblestone, brick, wood, grass, steel, or black. If left empty, the prop will have the a simple checkered gray material.
- **tag** (string) – The tag to apply to the prop. Useful for task references.

step(*action*, *ticks=1*, *publish=True*)

Supplies an action to the main agent and tells the environment to tick once. Primary mode of interaction for single agent environments.

Parameters

- **action** (np.ndarray) – An action for the main agent to carry out on the next tick.
- **ticks** (int) – Number of times to step the environment with this action. If ticks > 1, this function returns the last state generated.
- **publish** (bool) – Whether or not to publish as defined by scenario. Defaults to True.

Returns

A dictionary from agent name to its full state. The full state is another dictionary from holoocean.sensors.Sensors enum to np.ndarray, containing the sensors information for each sensor. The sensors always include the reward and terminal sensors. Reward and terminals can also be gotten through *get_reward_terminal()*.

Will return the state from the last tick executed.

Return type dict

tick(*num_ticks=1*, *publish=True*)

Ticks the environment once. Normally used for multi-agent environments.

Parameters

- **num_ticks** (int) – Number of ticks to perform. Defaults to 1.
- **publish** (bool) – Whether or not to publish as defined by scenario. Defaults to True.

Returns

A dictionary from agent name to its full state. The full state is another dictionary from holoocean.sensors.Sensors enum to np.ndarray, containing the sensors information for each sensor. The sensors always include the reward and terminal sensors. Reward and terminals can also be gotten through *get_reward_terminal()*.

Will return the state from the last tick executed.

Return type dict

CHAPTER

TEN

SPACES

Contains action space definitions

Classes:

ActionSpace(shape[, buffer_shape])	Abstract ActionSpace class.
ContinuousActionSpace(shape[, low, high, ...])	Action space that takes floating point inputs.
DiscreteActionSpace(shape, low, high[, ...])	Action space that takes integer inputs.

class holoocean.spaces.**ActionSpace**(*shape*, *buffer_shape=None*)

Abstract ActionSpace class.

Parameters

- **shape** (list of int) – The shape of data that should be input to step or tick.
- **buffer_shape** (list of int, optional) – The shape of the data that will be written to the shared memory.

  Only use this when it is different from shape.

Methods:

get_high()	The maximum value(s) for the action space.
get_low()	The minimum value(s) for the action space.
sample()	Sample from the action space.

Attributes:

shape	Get the shape of the action space.

get_high()

The maximum value(s) for the action space.

Returns the action space's maximum value(s)

Return type (list of float or float)

get_low()

The minimum value(s) for the action space.

Returns the action space's minimum value(s)

Return type (list of float or float)

sample()
Sample from the action space.

Returns A valid command to be input to step or tick.

Return type (np.ndarray)

property shape
Get the shape of the action space.

Returns The shape of the action space.

Return type (list of int)

class holoocean.spaces.**ContinuousActionSpace**(*shape*, *low=None*, *high=None*, *sample_fn=None*, *buffer_shape=None*)
Action space that takes floating point inputs.

Parameters

- **shape** (list of int) – The shape of data that should be input to step or tick.
- **sample_fn** (*function, optional*) – A function that takes a shape parameter and outputs a sampled command.
- **low** (list of float or float) – the low value(s) for the action space. Can be a scalar or an array
- **high** (list of float or float) – the high value(s) for the action space. Cand be a scalar or an array

  If this is not given, it will default to sampling from a unit gaussian.
- **buffer_shape** (list of int, optional) – The shape of the data that will be written to the shared memory.

  Only use this when it is different from shape.

Methods:

get_high()	The maximum value(s) for the action space.
get_low()	The minimum value(s) for the action space.
sample()	Sample from the action space.

get_high()
The maximum value(s) for the action space.

Returns the action space's maximum value(s)

Return type (list of float or float)

get_low()
The minimum value(s) for the action space.

Returns the action space's minimum value(s)

Return type (list of float or float)

sample()
Sample from the action space.

Returns A valid command to be input to step or tick.

Return type (np.ndarray)

class holoocean.spaces.**DiscreteActionSpace**(*shape*, *low*, *high*, *buffer_shape=None*)

Action space that takes integer inputs.

Parameters

- **shape** (list of int) – The shape of data that should be input to step or tick.
- **low** (int) – The lowest value to sample.
- **high** (int) – The highest value to sample.
- **buffer_shape** (list of int, optional) – The shape of the data that will be written to the shared memory.

  Only use this when it is different from shape.

Methods:

get_high()	The maximum value(s) for the action space.
get_low()	The minimum value(s) for the action space.
sample()	Sample from the action space.

get_high()

The maximum value(s) for the action space.

Returns the action space's maximum value(s)

Return type (list of float or float)

get_low()

The minimum value(s) for the action space.

Returns the action space's minimum value(s)

Return type (list of float or float)

sample()

Sample from the action space.

Returns A valid command to be input to step or tick.

Return type (np.ndarray)

CHAPTER

ELEVEN

COMMANDS

This module contains the classes used for formatting and sending commands to the HoloOcean backend. Most of these commands are just used internally by HoloOcean, regular users do not need to worry about these.

Classes:

AddSensorCommand(sensor_definition)	Add a sensor to an agent
Command()	Base class for Command objects.
CommandCenter(client)	Manages pending commands to send to the client (the engine).
CommandsGroup()	Represents a list of commands
CustomCommand(name[, num_params, string_params])	Send a custom command to the currently loaded world.
DebugDrawCommand(draw_type, start, end, ...)	Draw debug geometry in the world.
RGBCameraRateCommand(agent_name, ...)	Set the number of ticks between captures of the RGB camera.
RemoveSensorCommand(agent, sensor)	Remove a sensor from an agent
RenderQualityCommand(render_quality)	Adjust the rendering quality of HoloOcean
RenderViewportCommand(render_viewport)	Enable or disable the viewport.
RotateSensorCommand(agent, sensor, rotation)	Rotate a sensor on the agent
SendAcousticMessageCommand(from_agent_name, ...)	Set the number of ticks between captures of the RGB camera.
SendOpticalMessageCommand(from_agent_name, ...)	Send information through OpticalModem.
SpawnAgentCommand(location, rotation, name, ...)	Spawn an agent in the world.
TeleportCameraCommand(location, rotation)	Move the viewport camera (agent follower)

class holoocean.command.**AddSensorCommand**(*sensor_definition*)

Add a sensor to an agent

Parameters **sensor_definition** (SensorDefinition) – Sensor to add

class holoocean.command.**Command**

Base class for Command objects.

Commands are used for IPC between the holoocean python bindings and holoocean binaries.

Derived classes must set the `_command_type`.

The order in which *add_number_parameters()* and *add_number_parameters()* are called is significant, they are added to an ordered list. Ensure that you are adding parameters in the order the client expects them.

Methods:

add_number_parameters(number)	Add given number parameters to the internal list.
add_string_parameters(string)	Add given string parameters to the internal list.
set_command_type(command_type)	Set the type of the command.
to_json()	Converts to json.

add_number_parameters(*number*)
Add given number parameters to the internal list.

Parameters **number** (list of int/float, or singular int/float) – A number or list of numbers to add to the parameters.

add_string_parameters(*string*)
Add given string parameters to the internal list.

Parameters **string** (list of str or str) – A string or list of strings to add to the parameters.

set_command_type(*command_type*)
Set the type of the command.

Parameters **command_type** (str) – This is the name of the command that it will be set to.

to_json()
Converts to json.

Returns This object as a json string.

Return type str

class holoocean.command.**CommandCenter**(*client*)
Manages pending commands to send to the client (the engine).

Parameters **client** (*HoloOceanClient*) – Client to send commands to

Methods:

clear()	Clears pending commands
enqueue_command(command_to_send)	Adds command to outgoing queue.
handle_buffer()	Writes the list of commands into the command buffer, if needed.

Attributes:

queue_size	Returns: int: Size of commands queue

clear()
Clears pending commands

enqueue_command(*command_to_send*)
Adds command to outgoing queue.

Parameters **command_to_send** (*Command*) – Command to add to queue

handle_buffer()
Writes the list of commands into the command buffer, if needed.

Checks if we should write to the command buffer, writes all of the queued commands to the buffer, and then clears the contents of the self._commands list

property queue_size
Returns: int: Size of commands queue

class holoocean.command.**CommandsGroup**
Represents a list of commands

Can convert list of commands to json.

Methods:

add_command(command)	Adds a command to the list
clear()	Clear the list of commands.
to_json()	**returns** Json for commands array object and all of the commands inside the array.

Attributes:

size	Returns: int: Size of commands group

add_command(*command*)
Adds a command to the list

Parameters **command** (*Command*) – A command to add.

clear()
Clear the list of commands.

property size
Returns: int: Size of commands group

to_json()

Returns Json for commands array object and all of the commands inside the array.

Return type str

class holoocean.command.**CustomCommand**(*name*, *num_params=None*, *string_params=None*)
Send a custom command to the currently loaded world.

Parameters

- **name** (str) – The name of the command, ex "OpenDoor"
- **(obj** (*string_params*) – *list* of int): List of arbitrary number parameters
- **(obj** – *list* of int): List of arbitrary string parameters

class holoocean.command.**DebugDrawCommand**(*draw_type*, *start*, *end*, *color*, *thickness*, *lifetime*)
Draw debug geometry in the world.

Parameters

- **draw_type** (int) – The type of object to draw
  - 0: line
  - 1: arrow
  - 2: box
  - 3: point

- **start** (list of float) – The start [x, y, z] location in meters of the object. (see *Coordinate System*)
- **end** (list of float) – The end [x, y, z] location in meters of the object (not used for point, and extent for box)
- **color** (list of float) – [r, g, b] color value (from 0 to 255).
- **thickness** (float) – thickness of the line/object
- **lifetime** (float) – Number of simulation seconds the object should persist. If 0, makes persistent

class holoocean.command.**RGBCameraRateCommand**(*agent_name*, *sensor_name*, *ticks_per_capture*)
Set the number of ticks between captures of the RGB camera.

Parameters

- **agent_name** (str) – name of the agent to modify
- **sensor_name** (str) – name of the sensor to modify
- **ticks_per_capture** (int) – number of ticks between captures

class holoocean.command.**RemoveSensorCommand**(*agent*, *sensor*)
Remove a sensor from an agent

Parameters

- **agent** (str) – Name of agent to modify
- **sensor** (str) – Name of the sensor to remove

class holoocean.command.**RenderQualityCommand**(*render_quality*)
Adjust the rendering quality of HoloOcean

Parameters **render_quality** (*int*) – 0 = low, 1 = medium, 3 = high, 3 = epic

class holoocean.command.**RenderViewportCommand**(*render_viewport*)
Enable or disable the viewport. Note that this does not prevent the viewport from being shown, it just prevents it from being updated.

Parameters **render_viewport** (bool) – If viewport should be rendered

class holoocean.command.**RotateSensorCommand**(*agent*, *sensor*, *rotation*)
Rotate a sensor on the agent

Parameters

- **agent** (str) – Name of agent
- **sensor** (str) – Name of the sensor to rotate
- **rotation** (list of float) – [roll, pitch, yaw] rotation for sensor.

class holoocean.command.**SendAcousticMessageCommand**(*from_agent_name*, *from_sensor_name*, *to_agent_name*, *to_sensor_name*)
Set the number of ticks between captures of the RGB camera.

Parameters

- **agent_name** (str) – name of the agent to modify
- **sensor_name** (str) – name of the sensor to modify
- **num** (int) – number of ticks between captures

class holoocean.command.**SendOpticalMessageCommand**(*from_agent_name*, *from_sensor_name*, *to_agent_name*, *to_sensor_name*)

Send information through OpticalModem.

class holoocean.command.**SpawnAgentCommand**(*location*, *rotation*, *name*, *agent_type*, *is_main_agent=False*)

Spawn an agent in the world.

Parameters

- **location** (list of float) – [x, y, z] location to spawn agent (see *Coordinate System*)
- **name** (str) – The name of the agent.
- **agent_type** (str or type) – The type of agent to spawn (UAVAgent, NavAgent, ...)

Methods:

set_location(location)	Set where agent will be spawned.
set_name(name)	Set agents name
set_rotation(rotation)	Set where agent will be spawned.
set_type(agent_type)	Set the type of agent.

set_location(*location*)

Set where agent will be spawned.

Parameters **location** (list of float) – [x, y, z] location to spawn agent (see *Coordinate System*)

set_name(*name*)

Set agents name

Parameters **name** (str) – The name to set the agent to.

set_rotation(*rotation*)

Set where agent will be spawned.

Parameters **rotation** (list of float) – [roll, pitch, yaw] rotation for agent. (see *Rotations*)

set_type(*agent_type*)

Set the type of agent.

Parameters **agent_type** (str or type) – The type of agent to spawn.

class holoocean.command.**TeleportCameraCommand**(*location*, *rotation*)

Move the viewport camera (agent follower)

Parameters

- **location** (list of float) – The [x, y, z] location to give the camera (see *Coordinate System*)
- **rotation** (list of float) – The [roll, pitch, yaw] rotation to give the camera (see *Rotations*)

CHAPTER

TWELVE

HOLOOCEAN CLIENT

The client used for subscribing shared memory between python and c++.

Classes:

HoloOceanClient([uuid])	HoloOceanClient for controlling a shared memory session.

class holoocean.holooceanclient.**HoloOceanClient**(*uuid=''*)

HoloOceanClient for controlling a shared memory session.

Parameters **uuid** (str, optional) – A UUID to indicate which server this client is associated with. The same UUID should be passed to the world through a command line flag. Defaults to "".

Methods:

acquire([timeout])	Used to acquire control.
malloc(key, shape, dtype)	Allocates a block of shared memory, and returns a numpy array whose data corresponds with that block.
release()	Used to release control.

acquire(*timeout=10*)

Used to acquire control. Will wait until the HolodeckServer has finished its work.

malloc(*key*, *shape*, *dtype*)

Allocates a block of shared memory, and returns a numpy array whose data corresponds with that block.

Parameters

- **key** (str) – The key to identify the block.
- **shape** (list of int) – The shape of the numpy array to allocate.
- **dtype** (*type*) – The numpy data type (e.g. np.float32).

Returns The numpy array that is positioned on the shared memory.

Return type np.ndarray

release()

Used to release control. Will allow the HolodeckServer to take a step.

CHAPTER

THIRTEEN

PACKAGE MANAGER

Package manager for worlds available to download and use for HoloOcean

Functions:

available_packages()	Returns a list of package names available for the current version of HoloOcean
get_binary_path_for_package(package_name)	Gets the path to the binary of a specific package.
get_binary_path_for_scenario(scenario_name)	Gets the path to the binary for a given scenario name
get_package_config_for_scenario(scenario)	For the given scenario, returns the package config associated with it (config.json)
get_scenario(scenario_name)	Gets the scenario configuration associated with the given name
install(package_name[, url, branch, commit])	Installs a holoocean package.
installed_packages()	Returns a list of all installed packages
load_scenario_file(scenario_path)	Loads the scenario config file and returns a dictionary containing the configuration
package_info(pkg_name)	Prints the information of a package.
prune()	Prunes old versions of holoocean, other than the running version.
remove(package_name)	Removes a holoocean package.
remove_all_packages()	Removes all holoocean packages.
scenario_info([scenario_name, scenario, ...])	Gets and prints information for a particular scenario file Must match this format: scenario_name.json
world_info(world_name[, world_config, ...])	Gets and prints the information of a world.

holoocean.packagemanager.**available_packages**()
Returns a list of package names available for the current version of HoloOcean

Returns (list of str): List of package names

holoocean.packagemanager.**get_binary_path_for_package**(*package_name*)
Gets the path to the binary of a specific package.

Parameters **package_name** (str) – Name of the package to search for

Returns Returns the path to the config directory

Return type str

Raises *NotFoundException* – When the package requested is not found

holoocean.packagemanager.**get_binary_path_for_scenario**(*scenario_name*)
Gets the path to the binary for a given scenario name

Parameters **scenario_name** (str) – name of the configuration to load - eg "UrbanCity-Follow" Must be an exact match. Name must be unique among all installed packages

Returns A dictionary containing the configuration file

Return type dict

holoocean.packagemanager.**get_package_config_for_scenario**(*scenario*)
For the given scenario, returns the package config associated with it (config.json)

Parameters **scenario** (dict) – scenario dict to look up the package for

Returns package configuration dictionary

Return type dict

holoocean.packagemanager.**get_scenario**(*scenario_name*)
Gets the scenario configuration associated with the given name

Parameters **scenario_name** (str) – name of the configuration to load - eg "UrbanCity-Follow" Must be an exact match. Name must be unique among all installed packages

Returns A dictionary containing the configuration file

Return type dict

holoocean.packagemanager.**install**(*package_name*, *url=None*, *branch=None*, *commit=None*)
Installs a holoocean package.

Parameters **package_name** (str) – The name of the package to install

holoocean.packagemanager.**installed_packages**()
Returns a list of all installed packages

Returns List of all the currently installed packages

Return type list of str

holoocean.packagemanager.**load_scenario_file**(*scenario_path*)
Loads the scenario config file and returns a dictionary containing the configuration

Parameters **scenario_path** (str) – Path to the configuration file

Returns A dictionary containing the configuration file

Return type dict

holoocean.packagemanager.**package_info**(*pkg_name*)
Prints the information of a package.

Parameters **pkg_name** (str) – The name of the desired package to get information

holoocean.packagemanager.**prune**()
Prunes old versions of holoocean, other than the running version.

DO NOT USE WITH HOLODECKPATH

Don't use this function if you have overidden the path.

holoocean.packagemanager.**remove**(*package_name*)
Removes a holoocean package.

Parameters **package_name** (str) – the name of the package to remove

holoocean.packagemanager.**remove_all_packages**()
Removes all holoocean packages.

holoocean.packagemanager.**scenario_info**(*scenario_name=''*, *scenario=None*, *base_indent=0*)
Gets and prints information for a particular scenario file Must match this format: scenario_name.json

Parameters

- **scenario_name** (str) – The name of the scenario
- **scenario** (dict, optional) – Loaded dictionary config (overrides world_name and scenario_name)
- **base_indent** (int, optional) – How much to indent output by

holoocean.packagemanager.**world_info**(*world_name*, *world_config=None*, *base_indent=0*)
Gets and prints the information of a world.

Parameters

- **world_name** (str) – the name of the world to retrieve information for
- **world_config** (dict, optional) – A dictionary containing the world's configuration. Will find the config if None. Defaults to None.
- **base_indent** (int, optional) – How much to indent output

CHAPTER

FOURTEEN

SENSORS

Definition of all of the sensor information

Classes:

AcousticBeaconSensor(client, agent_name, ...)	Acoustic Beacon Sensor.
DVLSensor(client, agent_name, agent_type[, ...])	Doppler Velocity Log Sensor.
DepthSensor(client, agent_name, agent_type)	Pressure/Depth Sensor.
GPSSensor(client, agent_name, agent_type[, ...])	Gets the location of the agent in the world if the agent is close enough to the surface.
HoloOceanSensor(client[, agent_name, ...])	Base class for a sensor
IMUSensor(client, agent_name, agent_type[, ...])	Inertial Measurement Unit sensor.
ImagingSonar(client, agent_name, agent_type)	Simulates an imaging sonar.
LocationSensor(client, agent_name, agent_type)	Gets the location of the agent in the world.
OpticalModemSensor(client, agent_name, ...)	Handles communication between agents using an optical modem.
OrientationSensor(client[, agent_name, ...])	Gets the forward, right, and up vector for the agent.
PoseSensor(client[, agent_name, agent_type, ...])	Gets the forward, right, and up vector for the agent.
ProfilingSonar(client, agent_name, agent_type)	Simulates a multibeam profiling sonar.
RGBCamera(client, agent_name, agent_type[, ...])	Captures agent's view.
RangeFinderSensor(client, agent_name, agent_type)	Returns distances to nearest collisions in the directions specified by the parameters.
RotationSensor(client[, agent_name, ...])	Gets the rotation of the agent in the world, with rotation XYZ about the fixed frame, in degrees.
SensorDefinition(agent_name, agent_type, ...)	A class for new sensors and their parameters, to be used for adding new sensors.
SensorFactory()	Given a sensor definition, constructs the appropriate HoloOceanSensor object.
SidescanSonar(client, agent_name, agent_type)	Simulates a sidescan sonar.
SinglebeamSonar(client, agent_name, agent_type)	Simulates an echosounder, which is a sonar sensor with a single cone shaped beam.
VelocitySensor(client[, agent_name, ...])	Returns the x, y, and z velocity of the sensor in the global frame.
ViewportCapture(client, agent_name, agent_type)	Captures what the viewport is seeing.
WorldNumSensor(client[, agent_name, ...])	Returns any numeric value from the world corresponding to a given key.

class holoocean.sensors.**AcousticBeaconSensor**(*client*, *agent_name*, *agent_type*, *name='AcousticBeaconSensor'*, *config=None*)

Acoustic Beacon Sensor. Can send message to other beacon from the `send_acoustic_message()` command.

Returning array depends on sent message type. Note received message will be delayed due to time of acoustic

wave traveling. Possibly message types are, with representing the azimuth, elevation, r range, and d depth in water,

- `OWAY`: One way message that sends `["OWAY", from_sensor, payload]`
- `OWAYU`: One way message that sends `["OWAYU", from_sensor, payload, , ]`
- `MSG_REQ`: Requests a return message of MSG_RESP and sends `["MSG_REQ", from_sensor, payload]`
- `MSG_RESP`: Return message that sends `["MSG_RESP", from_sensor, payload]`
- `MSG_REQU`: Requests a return message of MSG_RESPU and sends `["MSG_REQU", from_sensor, payload, , ]`
- `MSG_RESPU`: Return message that sends `["MSG_RESPU", from_sensor, payload, , , r]`
- `MSG_REQX`: Requests a return message of MSG_RESPX and sends `["MSG_REQX", from_sensor, payload, , , d]`
- `MSG_RESPX`: Return message that sends `["MSG_RESPX", from_sensor, payload, , , r, d]`

These messages types are based on the Blueprint Subsea SeaTrac X150

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `id`: Id of this sensor. If not given, they are numbered sequentially.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor
`sensor_data`	Get the sensor data buffer

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

property sensor_data
Get the sensor data buffer

Returns Current sensor data

Return type `np.ndarray` of size `self.data_shape`

class holoocean.sensors.**DVLSensor**(*client*, *agent_name*, *agent_type*, *name='DVLSensor'*, *config=None*)
Doppler Velocity Log Sensor.

Returns a 1D numpy array of:

```
[velocity_x, velocity_y, velocity_z, range_x_forw, range_y_forw, range_x_back,␣
↪range_y_back]
```

With the range potentially not returning if `ReturnRange` is set to false.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `Elevation`: Angle of each acoustic beam off z-axis pointing down. Only used for noise/visualization. Defaults to 90 => horizontal.
- `DebugLines`: Whether to show lines of each beam. Defaults to false.
- `VelSigma/VelCov`: Covariance/Std to be applied to each beam velocity. Can be scalar, 4-vector or 4x4-matrix. Set one or the other. Defaults to 0 => no noise.
- `ReturnRange`: Boolean of whether range of beams should also be returned. Defaults to true.
- `MaxRange`: Maximum range that can be returned by the beams.
- `RangeSigma/RangeCov`: Covariance/Std to be applied to each beam range. Can be scalar, 4-vector or 4x4-matrix. Set one or the other. Defaults to 0 => no noise.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**DepthSensor**(*client*, *agent_name*, *agent_type*, *name='DepthSensor'*, *config=None*)
Pressure/Depth Sensor.

Returns a 1D numpy array of:

```
[position_z]
```

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `Sigma/Cov`: Covariance/Std to be applied, a scalar. Defaults to 0 => no noise.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**GPSSensor**(*client*, *agent_name*, *agent_type*, *name='GPSSensor'*, *config=None*)
Gets the location of the agent in the world if the agent is close enough to the surface.

Returns coordinates in `[x, y, z]` format (see *Coordinate System*)

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `Sigma/Cov`: Covariance/Std of measurement. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.
- `Depth`: How deep in the water we can still receive GPS messages in meters. Defaults to 2m.
- `DepthSigma/DepthCov`: Covariance/Std of depth. Must be a scalar. Set one or the other. Defaults to 0 => no noise.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor
`sensor_data`	Get the sensor data buffer

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

property sensor_data
Get the sensor data buffer

Returns Current sensor data

Return type `np.ndarray` of size `self.data_shape`

class holoocean.sensors.**HoloOceanSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)
Base class for a sensor

Parameters

- **client** (*HoloOceanClient*) – Client attached to a sensor
- **agent_name** (`str`) – Name of the parent agent
- **agent_type** (`str`) – Type of the parent agent
- **name** (`str`) – Name of the sensor
- **config** (`dict`) – Configuration dictionary to pass to the engine

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor
`sensor_data`	Get the sensor data buffer

Methods:

`rotate`(rotation)	Rotate the sensor.

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

rotate(*rotation*)
Rotate the sensor. It will be applied in approximately three ticks. *step()* or *tick()*.)

This will not persist after a call to reset(). If you want a persistent rotation for a sensor, specify it in your scenario configuration.

Parameters **rotation** (`list` of `float`) – rotation for sensor (see *Rotations*).

property sensor_data
Get the sensor data buffer

Returns Current sensor data

Return type `np.ndarray` of size `self.data_shape`

class holoocean.sensors.**IMUSensor**(*client*, *agent_name*, *agent_type*, *name='IMUSensor'*, *config=None*)
Inertial Measurement Unit sensor.

Returns a 2D numpy array of:

```
[ [accel_x, accel_y, accel_z],
  [ang_vel_roll,  ang_vel_pitch, ang_vel_yaw],
  [accel_bias_x, accel_bias_y, accel_bias_z],
  [ang_vel_bias_roll,  ang_vel_bias_pitch, ang_vel_bias_yaw]    ]
```

where the accleration components are in m/s and the angular velocity is in rad/s.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `AccelSigma/AccelCov`: Covariance/Std for acceleration component. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.
- `AngVelSigma/AngVelCov`: Covariance/Std for angular velocity component. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.

- `AccelBiasSigma/AccelCBiasov`: Covariance/Std for acceleration bias component. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.
- `AngVelBiasSigma/AngVelBiasCov`: Covariance/Std for acceleration bias component. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.
- `ReturnBias`: Whether the sensor should return the bias along with accel/ang. vel. Defaults to false.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**ImagingSonar**(*client*, *agent_name*, *agent_type*, *name='ImagingSonar'*, *config=None*)

Simulates an imaging sonar. See *Configuring Octree* for more on how to configure the octree that is used.

The `configuration` block (see *Configuration Block*) accepts any of the options in the following sections.

Basic Configuration

- `Azimuth`: Azimuth (side to side) angle visible in degrees, defaults to 120.
- `Elevation`: Elevation angle (up and down) visible in degrees, defaults to 20.
- `RangeMin`: Minimum range visible in meters, defaults to 0.1.
- `RangeMax`: Maximum range visible in meters, defaults to 10.
- `RangeBins/RangeRes`: Number of range bins of resulting image, or resolution (length in meters) of each bin. Set one or the other. Defaults to 512 bins.
- `AzimuthBins/AzimuthRes`: Number of azimuth bins of resulting image, or resolution (length in degrees) of each bin. Set one or the other. Defaults to 512 bins.

Noise Configuration

- `AddSigma/AddCov`: Additive noise std/covariance from a Rayleigh distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultSigma/MultCov`: Multiplication noise std/covariance from a normal distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultiPath`: Whether to compute multipath or not. Defaults to False.
- `ClusterSize`: Size of cluster when multipath is enabled. Defaults to 5.
- `ScaleNoise`: Whether to scale the returned intensities or not. Defaults to False.
- `AzimuthStreaks`: What sort of azimuth artifacts to introduce. -1 is a removal artifact, 0 is no artifact, and 1 is increased gain artifact. Defaults to 0.

- `RangeSigma`: Additive noise std from an exponential distribution that will be added to the range measurements, and the intensities will be scaled by the pdf. Needs to be a float. Defaults to 0, or off.

Advanced Configuration

- `ShowWarning`: Whether to show on screen warning about sonar computation happening. Defaults to True.
- `ElevationBins/ElevationRes`: Number of elevation bins used when shadowing is done, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the octree size and the min/max range. Should only be set if shadowing isn't working.
- `InitOctreeRange`: Upon startup, all mid-level octrees within this distance of the agent will be created.
- `ViewRegion`: Turns on green lines to see visible region. Defaults to False.
- `ViewOctree`: What octree leaves to show. Less than -1 means none, -1 means all, and anything greater than or equal to 0 shows the corresponding beam index. Defaults to -10.
- `ShadowEpsilon`: What constitutes a break between clusters when shadowing. Defaults to 4*OctreeMin.
- `WaterDensity`: Density of water in kg/m^3. Defaults to 997.
- `WaterSpeedSound`: Speed of sound in water in m/s. Defaults to 1480.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**LocationSensor**(*client*, *agent_name*, *agent_type*, *name='LocationSensor'*, *config=None*)

Gets the location of the agent in the world.

Returns coordinates in `[x, y, z]` format (see *Coordinate System*)

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `Sigma/Cov`: Covariance/Std. Can be scalar, 3-vector or 3x3-matrix. Set one or the other. Defaults to 0 => no noise.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**OpticalModemSensor**(*client*, *agent_name*, *agent_type*, *name='OpticalModemSensor'*, *config=None*)
Handles communication between agents using an optical modem. Can send message to other modem from the `send_optical_message()` command.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `MaxDistance`: Max Distance in meters of OpticalModem. (default 50)
- `id`: Id of this sensor. If not given, they are numbered sequentially.
- `DistanceSigma/DistanceCov`: Determines the standard deviation/covariance of the noise on MaxDistance. Must be scalar value. (default 0 => no noise)
- `AngleSigma/AngleCov`: Determines the standard deviation of the noise on LaserAngle. Must be scalar value. (default 0 => no noise)
- `LaserDebug`: Show debug traces. (default false)
- `DebugNumSides`: Number of sides on the debug cone. (default 72)
- `LaserAngle`: Angle of lasers from origin. Measured in degrees. (default 60)

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor
`sensor_data`	Get the sensor data buffer

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

property sensor_data
Get the sensor data buffer

Returns Current sensor data

Return type `np.ndarray` of size `self.data_shape`

class holoocean.sensors.**OrientationSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Gets the forward, right, and up vector for the agent. Returns a 2D numpy array of

```
[ [forward_x, right_x, up_x],
  [forward_y, right_y, up_y],
  [forward_z, right_z, up_z] ]
```

Attributes:

data_shape	The shape of the sensor data
dtype	The type of data in the sensor

property data_shape

The shape of the sensor data

Returns Sensor data shape

Return type tuple

property dtype

The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**PoseSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Gets the forward, right, and up vector for the agent. Returns a 2D numpy array of

```
[ [R, p],
  [0, 1] ]
```

where R is the rotation matrix (See OrientationSensor) and p is the robot world location (see LocationSensor)

Attributes:

data_shape	The shape of the sensor data
dtype	The type of data in the sensor

property data_shape

The shape of the sensor data

Returns Sensor data shape

Return type tuple

property dtype

The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**ProfilingSonar**(*client*, *agent_name*, *agent_type*, *name='ProfilingSonar'*, *config=None*)

Simulates a multibeam profiling sonar. This is largely based off of the imaging sonar (*ImagingSonar*), just with different defaults. See *Configuring Octree* for more on how to configure the octree that is used.

The `configuration` block (see *Configuration Block*) accepts any of the options in the following sections.

Basic Configuration

- `Azimuth`: Azimuth (side to side) angle visible in degrees, defaults to 120.
- `Elevation`: Elevation angle (up and down) visible in degrees, defaults to 1.
- `RangeMin`: Minimum range visible in meters, defaults to 0.5.
- `RangeMax`: Maximum range visible in meters, defaults to 75.
- `RangeBins/RangeRes`: Number of range bins of resulting image, or resolution (length in meters) of each bin. Set one or the other. Defaults to 750 bins.
- `AzimuthBins/AzimuthRes`: Number of azimuth bins of resulting image, or resolution (length in degrees) of each bin. Set one or the other. Defaults to 480 bins.

Noise Configuration

- `AddSigma/AddCov`: Additive noise std/covariance from a Rayleigh distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultSigma/MultCov`: Multiplication noise std/covariance from a normal distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultiPath`: Whether to compute multipath or not. Defaults to False.
- `ClusterSize`: Size of cluster when multipath is enabled. Defaults to 5.
- `ScaleNoise`: Whether to scale the returned intensities or not. Defaults to False.
- `AzimuthStreaks`: What sort of azimuth artifacts to introduce. -1 is a removal artifact, 0 is no artifact, and 1 is increased gain artifact. Defaults to 0.

Advanced Configuration

- `ShowWarning`: Whether to show on screen warning about sonar computation happening. Defaults to True.
- `ElevationBins/ElevationRes`: Number of elevation bins used when shadowing is done, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the octree size and the min/max range. Should only be set if shadowing isn't working.
- `InitOctreeRange`: Upon startup, all mid-level octrees within this distance of the agent will be created.
- `ViewRegion`: Turns on green lines to see visible region. Defaults to False.
- `ViewOctree`: What octree leaves to show. Less than -1 means none, -1 means all, and anything greater than or equal to 0 shows the corresponding beam index. Defaults to -10.
- `ShadowEpsilon`: What constitutes a break between clusters when shadowing. Defaults to 4*OctreeMin.
- `WaterDensity`: Density of water in kg/m^3. Defaults to 997.
- `WaterSpeedSound`: Speed of sound in water in m/s. Defaults to 1480.

`class holoocean.sensors.RGBCamera`(*client*, *agent_name*, *agent_type*, *name='RGBCamera'*, *config=None*)

Captures agent's view.

The default capture resolution is 256x256x256x4, corresponding to the RGBA channels. The resolution can be increased, but will significantly impact performance.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `CaptureWidth`: Width of captured image

- `CaptureHeight`: Height of captured image

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

Methods:

`set_ticks_per_capture`(ticks_per_capture)	Sets this RGBCamera to capture a new frame every ticks_per_capture.

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

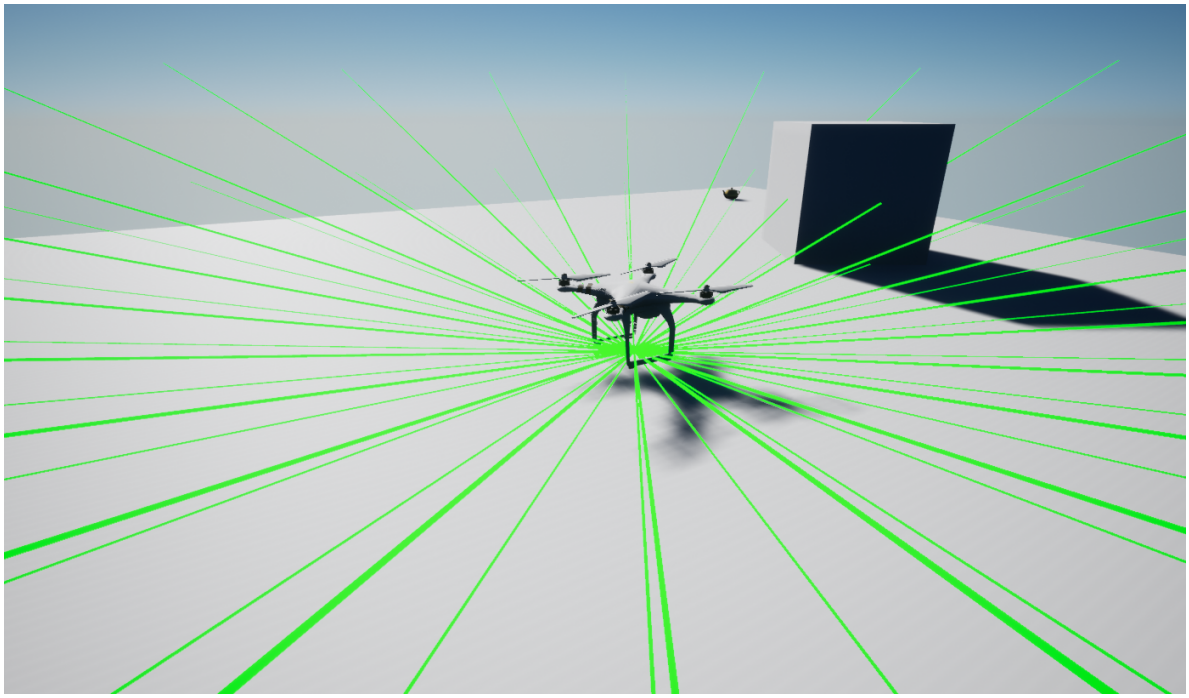
set_ticks_per_capture(*ticks_per_capture*)
Sets this RGBCamera to capture a new frame every ticks_per_capture.

The sensor's image will remain unchanged between captures.

This method must be called after every call to env.reset.

Parameters **ticks_per_capture** (`int`) – The amount of ticks to wait between camera captures.

class holoocean.sensors.**RangeFinderSensor**(*client*, *agent_name*, *agent_type*, *name='RangeFinderSensor'*, *config=None*)
Returns distances to nearest collisions in the directions specified by the parameters. For example, if an agent had two range sensors at different angles with 24 lasers each, the LaserDebug traces would look something like this:



Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `LaserMaxDistance`: Max Distance in meters of RangeFinder. (default 10)
- `LaserCount`: Number of lasers in sensor. (default 1)
- `LaserAngle`: Angle of lasers from origin. Measured in degrees. Positive angles point up. (default 0)
- `LaserDebug`: Show debug traces. (default false)

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**RotationSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Gets the rotation of the agent in the world, with rotation XYZ about the fixed frame, in degrees.

Returns `[roll, pitch, yaw]` (see *Rotations*)

Attributes:

data_shape	The shape of the sensor data
dtype	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type tuple

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**SensorDefinition**(*agent_name*, *agent_type*, *sensor_name*, *sensor_type*, *socket=''*, *location=(0, 0, 0)*, *rotation=(0, 0, 0)*, *config=None*, *existing=False*, *lcm_channel=None*, *tick_every=None*)
A class for new sensors and their parameters, to be used for adding new sensors.

Parameters

- **agent_name** (str) – The name of the parent agent.
- **agent_type** (str) – The type of the parent agent
- **sensor_name** (str) – The name of the sensor.
- **sensor_type** (str or *HoloOceanSensor*) – The type of the sensor.
- **socket** (str, optional) – The name of the socket to attach sensor to.
- **location** (Tuple of float, optional) – [x, y, z] coordinates to place sensor relative to agent (or socket) (see *Coordinate System*).
- **rotation** (Tuple of float, optional) – [roll, pitch, yaw] to rotate sensor relative to agent (see *Rotations*)
- **config** (dict) – Configuration dictionary for the sensor, to pass to engine

Methods:

get_config_json_string()	Gets the configuration dictionary as a string ready for transport

get_config_json_string()
Gets the configuration dictionary as a string ready for transport

Returns The configuration as an escaped json string

Return type (str)

class holoocean.sensors.**SensorFactory**
Given a sensor definition, constructs the appropriate HoloOceanSensor object.

Methods:

build_sensor(client, sensor_def)	Constructs a given sensor associated with client

static build_sensor(*client*, *sensor_def*)
Constructs a given sensor associated with client

Parameters

- **client** (str) – Name of the agent this sensor is attached to
- **sensor_def** (*SensorDefinition*) – Sensor definition to construct

Returns:

class holoocean.sensors.**SidescanSonar**(*client*, *agent_name*, *agent_type*, *name='SidescanSonar'*, *config=None*)

Simulates a sidescan sonar. See *Configuring Octree* for more on how to configure the octree that is used.

The `configuration` block (see *Configuration Block*) accepts any of the options in the following sections.

Basic Configuration

- `Azimuth`: Azimuth (side to side) angle visible in degrees, defaults to 170.
- `Elevation`: Elevation angle (up and down) visible in degrees, defaults to 0.25.
- `RangeMin`: Minimum range visible in meters, defaults to 0.5.
- `RangeMax`: Maximum range visible in meters, defaults to 35.
- `RangeBins/RangeRes`: Number of range bins of resulting image, or resolution (length in meters) of each bin. Set one or the other. Defaults to 0.05 m.

Noise Configuration

- `AddSigma/AddCov`: Additive noise std/covariance from a Rayleigh distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultSigma/MultCov`: Multiplication noise std/covariance from a normal distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.

Advanced Configuration

- `ShowWarning`: Whether to show on screen warning about sonar computation happening. Defaults to True.
- `AzimuthBins/AzimuthRes`: Number of azimuth bins of resulting image, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the OctreeMin.
- `ElevationBins/ElevationRes`: Number of elevation bins used when shadowing is done, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the octree size and the min range. Should only be set if shadowing isn't working.
- `InitOctreeRange`: Upon startup, all mid-level octrees within this distance of the agent will be created.
- `ViewRegion`: Turns on green lines to see visible region. Defaults to False.
- `ViewOctree`: What octree leaves to show. Less than -1 means none, -1 means all, and anything greater than or equal to 0 shows the corresponding beam index. Defaults to -10.
- `ShadowEpsilon`: What constitutes a break between clusters when shadowing. Defaults to 4*OctreeMin.
- `WaterDensity`: Density of water in kg/m^3. Defaults to 997.
- `WaterSpeedSound`: Speed of sound in water in m/s. Defaults to 1480.

Attributes:

data_shape	The shape of the sensor data
dtype	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type tuple

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**SinglebeamSonar**(*client*, *agent_name*, *agent_type*, *name='SinglebeamSonar'*, *config=None*)

Simulates an echosounder, which is a sonar sensor with a single cone shaped beam. See *Configuring Octree* for more on how to configure the octree that is used.

Returns a 1D numpy array of the average intensities held in each range bin of the sensor. The length of the array is specified by the number of range bins chosen for the sensor.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

The `configuration` block (see *Configuration Block*) accepts any of the options in the following sections.

Basic Configuration

- `OpeningAngle`: Opening angle of the cone visible in degrees, defaults to 30. In this documentation, the opening angle would be 2 times the semi-vertical angle of the cone.
- `RangeMin`: Minimum range visible in meters, defaults to 0.5.
- `RangeMax`: Maximum range visible in meters, defaults to 10.
- `RangeBins/RangeRes`: Number of range bins of resulting image, or resolution (length in meters) of each bin. Set one or the other. Defaults to 200 bins.

Noise Configuration

- `AddSigma/AddCov`: Additive noise std/covariance from a Rayleigh distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `MultSigma/MultCov`: Multiplication noise std/covariance from a normal distribution. Needs to be a float. Set one or the other. Defaults to 0, or off.
- `RangeSigma`: Additive noise std from an exponential distribution that will be added to the range measurements. Needs to be a float. Defaults to 0/off.

Advanced Configuration

- `ShowWarning`: Whether to show on screen warning about sonar computation happening. Defaults to True.
- `OpeningAngleBins/OpeningAngleRes`: Number of OpeningAngle bins used when shadowing is done, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the octree size and the min/max range. Should only be set if shadowing isn't working.
- `CentralAngleBins/CentralAngleRes`: Number of CentralAngle bins used when shadowing is done, or resolution (length in degrees) of each bin. Set one or the other. By default this is computed based on the octree size and the min/max range. Should only be set if shadowing isn't working.
- `InitOctreeRange`: Upon startup, all mid-level octrees within this distance of the agent will be created.
- `ViewRegion`: Turns on green lines to see visible region. Defaults to False.

- `ViewOctree`: What octree leaves to show. Less than -1 means none, -1 means all, and anything greater than or equal to 0 shows the corresponding beam index. Defaults to -10.
- `ShadowEpsilon`: What constitutes a break between clusters when shadowing. Defaults to 4*OctreeMin.
- `WaterDensity`: Density of water in kg/m^3. Defaults to 997.
- `WaterSpeedSound`: Speed of sound in water in m/s. Defaults to 1480.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**VelocitySensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Returns the x, y, and z velocity of the sensor in the global frame.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**ViewportCapture**(*client*, *agent_name*, *agent_type*, *name='ViewportCapture'*, *config=None*)

Captures what the viewport is seeing.

The ViewportCapture is faster than the RGB camera, but there can only be one camera and it must capture what the viewport is capturing. If performance is critical, consider this camera instead of the RGBCamera.

It may be useful to position the camera with `teleport_camera()`.

Configuration

The `configuration` block (see *Configuration Block*) accepts the following options:

- `CaptureWidth`: Width of captured image
- `CaptureHeight`: Height of captured image

THESE DIMENSIONS MUST MATCH THE VIEWPORT DIMENSTIONS

If you have configured the size of the viewport (`window_height/width`), you must make sure that `CaptureWidth/Height` of this configuration block is set to the same dimensions.

The default resolution is `1280x720`, matching the default Viewport resolution.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

class holoocean.sensors.**WorldNumSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Returns any numeric value from the world corresponding to a given key. This is world specific.

Attributes:

`data_shape`	The shape of the sensor data
`dtype`	The type of data in the sensor

property data_shape
The shape of the sensor data

Returns Sensor data shape

Return type `tuple`

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

CHAPTER

FIFTEEN

LCM

LCM package __init__.py file This file automatically generated by lcm-gen. DO NOT MODIFY BY HAND!!!!

Classes:

SensorData(sensor_type, channel)	Wrapper class for the various types of publishable sensor data.

Functions:

gen(lang[, path, headers])	Generates LCM files for sensors in whatever language requested.

class holoocean.lcm.**SensorData**(*sensor_type*, *channel*)

Wrapper class for the various types of publishable sensor data.

Parameters

- **sensor_type** (str) – Type of sensor to be imported
- **channel** (str) – Name of channel to publish to.

Methods:

set_value(timestamp, value)	Set value in respective sensor class.

set_value(*timestamp*, *value*)

Set value in respective sensor class.

Parameters

- **timestamp** (int) – Number of milliseconds since last data was published
- **value** (list) – List of sensor data to put into LCM sensor class

holoocean.lcm.**gen**(*lang*, *path='.'*, *headers=None*)

Generates LCM files for sensors in whatever language requested.

Parameters

- **lang** (str) – One of “cpp”, “c”, “java”, “python”, “lua”, “csharp”, “go”
- **path** (str, optional) – Location to save files in. Defaults to current directory.
- **headers** (str, optional) – Where to store .h files for C . Defaults to same as c files, given by path arg.

CHAPTER

SIXTEEN

SHARED MEMORY

Shared memory with memory mapping

Classes:

Shmem(name, shape[, dtype, uuid])	Implementation of shared memory

class holoocean.shmem.**Shmem**(*name*, *shape*, *dtype=<class 'numpy.float32'>*, *uuid=''*)

Implementation of shared memory

Parameters

- **name** (str) – Name the points to the beginning of the shared memory block
- **shape** (int) – Shape of the memory block
- **dtype** (*type, optional*) – data type of the shared memory. Defaults to np.float32
- **uuid** (str, optional) – UUID of the memory block. Defaults to “”

Methods:

unlink()	unlinks the shared memory

unlink()

unlinks the shared memory

CHAPTER

SEVENTEEN

UTIL

Helpful Utilities

Functions:

convert_unicode(value)	Resolves python 2 issue with json loading in unicode instead of string
get_holoocean_path()	Gets the path of the holoocean environment
get_holoocean_version()	Gets the current version of holoocean
get_os_key()	Gets the key for the OS.
human_readable_size(size_bytes)	Gets a number of bytes as a human readable string.

holoocean.util.**convert_unicode**(*value*)
Resolves python 2 issue with json loading in unicode instead of string

Parameters **value** (str) – Unicode value to be converted

Returns Converted string

Return type (str)

holoocean.util.**get_holoocean_path**()
Gets the path of the holoocean environment

Returns path to the current holoocean environment

Return type (str)

holoocean.util.**get_holoocean_version**()
Gets the current version of holoocean

Returns the current version

Return type (str)

holoocean.util.**get_os_key**()
Gets the key for the OS.

Returns Linux or Windows. Throws NotImplementedError for other systems.

Return type str

holoocean.util.**human_readable_size**(*size_bytes*)
Gets a number of bytes as a human readable string.

Parameters **size_bytes** (int) – The number of bytes to get as human readable.

Returns The number of bytes in a human readable form.

Return type str

CHAPTER

EIGHTEEN

EXCEPTIONS

HoloOcean Exceptions

Exceptions:

HoloOceanConfigurationException	The user provided an invalid configuration for HoloOcean
HoloOceanException	Base class for a generic exception in HoloOcean.
NotFoundException	Raised when a package cannot be found
TimeoutException	Exception raised when communicating with the engine timed out.

exception holoocean.exceptions.**HoloOceanConfigurationException**
The user provided an invalid configuration for HoloOcean

exception holoocean.exceptions.**HoloOceanException**
Base class for a generic exception in HoloOcean.

Parameters **message** (*str*) – The error string.

exception holoocean.exceptions.**NotFoundException**
Raised when a package cannot be found

exception holoocean.exceptions.**TimeoutException**
Exception raised when communicating with the engine timed out.

CHAPTER

NINETEEN

WEATHER CONTROLLER

Weather/time controller for environments

Classes:

WeatherController(send_world_command)	Controller for dynamically changing weather and time in an environment

class holoocean.weather.**WeatherController**(*send_world_command*)

Controller for dynamically changing weather and time in an environment

Parameters **send_world_command** (*function*) – Callback for sending commands to a world

Methods:

set_day_time(hour)	Change the time of day.
set_fog_density(density)	Change the fog density.
set_weather(weather_type)	Set the world's weather.
start_day_cycle(day_length)	Start the day cycle.
stop_day_cycle()	Stop the day cycle.

set_day_time(*hour*)

Change the time of day.

Daytime will change when `tick()` or `step()` is called next.

By the next tick, the lighting and the skysphere will be updated with the new hour.

If there is no skysphere, skylight, or directional source light in the world, this command will exit the environment.

Parameters **hour** (`int`) – The hour in 24-hour format: [0, 23].

set_fog_density(*density*)

Change the fog density.

The change will occur when `tick()` or `step()` is called next.

By the next tick, the exponential height fog in the world will have the new density. If there is no fog in the world, it will be created with the given density.

Parameters **density** (`float`) – The new density value, between 0 and 1. The command will not be sent if the given density is invalid.

set_weather(*weather_type*)

Set the world's weather.

The new weather will be applied when `tick()` or `step()` is called next.

By the next tick, the lighting, skysphere, fog, and relevant particle systems will be updated and/or spawned to the given weather.

If there is no skysphere, skylight, or directional source light in the world, this command will exit the environment.

Note: Because this command can affect the fog density, any changes made by a `change_fog_density` command before a set_weather command called will be undone. It is recommended to call `change_fog_density` after calling set weather if you wish to apply your specific changes.

In all downloadable worlds, the weather is sunny by default.

If the given type string is not available, the command will not be sent.

Parameters

- **weather_type** (`str`) – The type of weather, which can be `rain`, `cloudy`, or
- **sunny.** –

start_day_cycle(*day_length*)
Start the day cycle.

The cycle will start when `tick()` or `step()` is called next.

The sky sphere will then update each tick with an updated sun angle as it moves about the sky. The length of a day will be roughly equivalent to the number of minutes given.

If there is no skysphere, skylight, or directional source light in the world, this command will exit the environment.

Parameters **day_length** (`int`) – The number of minutes each day will be.

stop_day_cycle()
Stop the day cycle.

The cycle will stop when `tick()` or `step()` is called next.

By the next tick, day cycle will stop where it is.

If there is no skysphere, skylight, or directional source light in the world, this command will exit the environment.

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

E

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W