Holodeck Documentation

Release 0.3.1

Joshua Greaves, Max Robinson, Nick Walton

Jan 26, 2021

HOLODECK DOCUMENTATION

Note: **Have a question?** Join our Discord!

CHAPTER

ONE

INSTALLATION

Holodeck is installed in two portions: a client python library (`holodeck`) 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 3.5
- Several gigabytes of storage
- pip3
- Linux: OpenGL 3+

1.2 Install Client via pip

The latest stable Holodeck package is available in a pip repository:

```
pip install holodeck
```

Note: On some Ubuntu systems a dependency of Holodeck (`posix-ipc`) can fail to install if you do not have the `python3-dev` package installed.

```
$ apt install python3-dev
```

1.3 Install Client via git

To use the latest version of Holodeck, you can install and use Holodeck simply by cloning the BYU-PCCL/holodeck repository, 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.

If you want to download a specific release of Holodeck, each release is tagged in the Git repository.

1.4 Docker Installation

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

You will need `nvidia-docker` installed.

The repository on DockerHub is pccl/holodeck.

Currently the following tags are availible:

- `base` : base image without any worlds
- `default-worlds` : comes with the default worlds pre-installed
- `dexterity` : comes with the dexterity package pre-installed

This is an example command to start a holodeck container

```
nvidia-docker run --rm -it --name holodeck pccl/holodeck:default-worlds
```

Note: Holodeck cannot be run with root privileges, so the user `holodeckuser` 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 holodeck import packagemanager
>>> packagemanager.installed_packages()
[]
>>> packagemanager.available_packages()
{'DefaultWorlds': ['0.1.0', '0.1.1'], 'MoveBox': ['0.0.1']}
>>> packagemanager.install("DefaultWorlds")
Installing DefaultWorlds ver. 0.1.1 from http://localhost:8080/packages/0.2.0/
↪DefaultWorlds/Linux/0.1.1.zip
File size: 1.55 GB
|| 100%
Unpacking worlds...
Finished.
>>> packagemanager.installed_packages()
['DefaultWorlds']
```

1.5.2 Installation Location

By default, Holodeck 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 Holodeck 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("DefaultWorlds")
Package: DefaultWorlds
   Platform: Linux
   Version: 1.04
   Path: LinuxNoEditor/Holodeck/Binaries/Linux/Holodeck
   Worlds:
   UrbanCity
         Scenarios:
         UrbanCity-Follow:
            Agents:
               Name: ThisIsAScenario
               Type: UavAgent
               Sensors:
               RGBCamera
               OrientationSensor
               LocationSensor
   CyberPunkCity
         Scenarios:
```

(continues on next page)

(continued from previous page)

```
        CyberPunkCity-Follow:
           Agents:
              Name: ThisIsAScenario
              Type: UavAgent
              Sensors:
              RGBCamera
              OrientationSensor
              LocationSensor
```

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

```
packagemanager.world_info("UrbanCity")
packagemanager.scenario_info("UrbanCity-Follow")
```

CHAPTER

TWO

GETTING STARTED

First, see *Installation* to get the `holodeck` package and `DefaultWorlds` installed.

A minimal Holodeck usage example is below:

```
import holodeck
import numpy as np

env = holodeck.make("UrbanCity-MaxDistance")

# The UAV takes 3 torques and a thrust as a command.
command = np.array([0, 0, 0, 100])

env.reset()
for _ in range(180):
    state, reward, terminal, info = env.step(command)
```

Notice that:

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

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

2. The interface of Holodeck is designed to be familiar to OpenAI Gym
3. You must call `.reset()` before calling `.step()` or `.tick()`

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

```
location_data = state["LocationSensor"]
```

That's it! Holodeck 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*.

2.1 Code Examples

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

2.1.1 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 holodeck, cv2

env = holodeck.make("MazeWorld-FinishMazeSphere")
env.act('sphere0', [0])

for _ in range(10):
    state = env.tick()

    pixels = state['sphere0'][holodeck.sensors.RGBCamera.sensor_type]
    cv2.namedWindow("Camera Output")
    cv2.moveWindow("Camera Output", 500, 500)
    cv2.imshow("Camera Output", pixels[:, :, 0:3])
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

2.1.2 Custom Scenario Configurations

Holodeck worlds are meant to be configurable by changing out the scenario (see *Scenarios*). There are some scenarios included with Holodeck packages distributed as `.json` files, but Holodeck 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*

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 `holodeck.make()`.

Example

```
import holodeck

cfg = {
    "name": "test_rgb_camera",
    "world": "ExampleWorld",
    "package_name": "DefaultWorlds",
    "main_agent": "sphere0",
    "agents": [
        {
            "agent_name": "sphere0",
            "agent_type": "SphereAgent",
            "sensors": [
                {
                    "sensor_type": "RGBCamera",
                    "socket": "CameraSocket",
```

(continues on next page)

(continued from previous page)

```
                    "configuration": {
                        "CaptureWidth": 512,
                        "CaptureHeight": 512
                    }
                }
            ],
            "control_scheme": 0,
            "location": [0, 0, 0]
        }
    ]
}

with holodeck.make(scenario_cfg=cfg) as env:
    env.tick()
```

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 Holodeck'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*

Holodeck's Scenario Search Path

When you give a scenario name to `holodeck.make()`, Holodeck 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 Holodeck will automatically find and use it.

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

2.1.3 Multi Agent Example

With Holodeck, 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.

You can access the reward, terminal and location for the UAV as shown below.

Code

```
import holodeck
import numpy as np

env = holodeck.make('CyberPunkCity-Follow')
env.reset()

env.act('uav0', np.array([0, 0, 0, 100]))
env.act('nav0', np.array([0, 0, 0]))
for i in range(300):
    states = env.tick()

    # states is a dictionary
    task = states["uav0"]["FollowTask"]

    reward = task[0]
    terminal = task[1]
    location = states["uav0"]["LocationSensor"]
```

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

CHAPTER

THREE

USING HOLODECK

3.1 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.

3.1.1 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.

3.2 Units and Coordinates in Holodeck

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

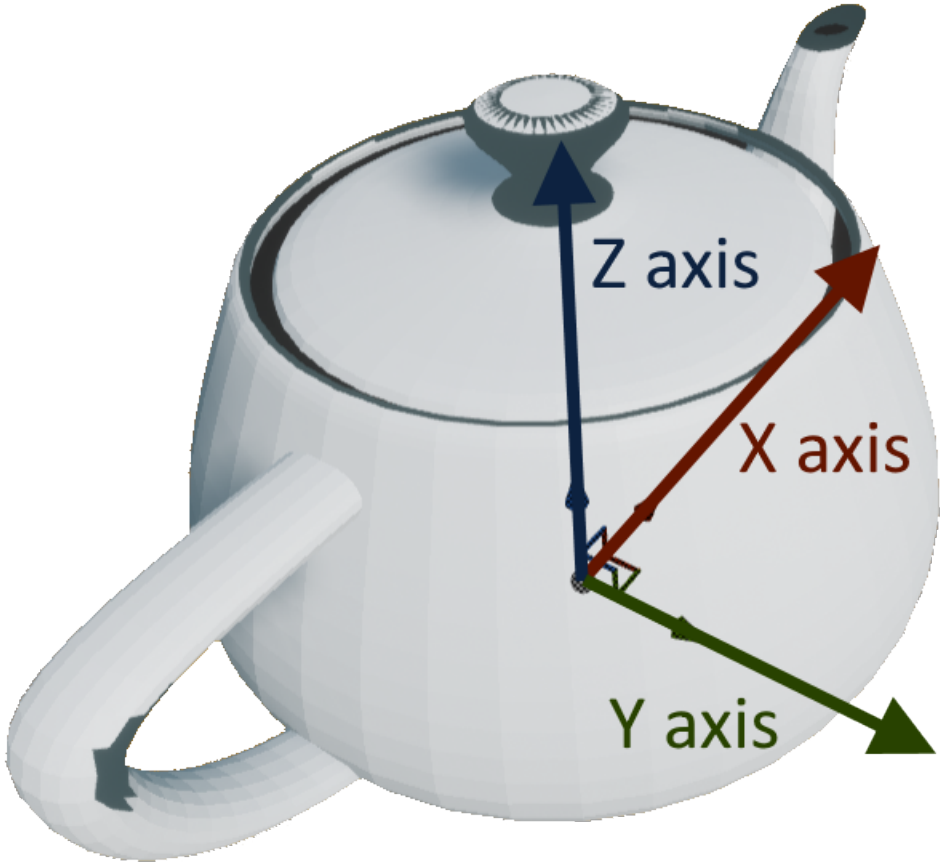
3.2.1 Coordinate System

Since Holodeck depends on Unreal Engine, we use a left handed coordinate system with positive `z` being up. This is something baked deep into the engine that we can't easily change.

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

`[x, y, z]` where:

- Positive `x` is **forward**
- Positive `y` is **right**
- 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).

3.2.2 Rotations

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

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

(source)

3.3 Improving Holodeck Performance

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

3.3.1 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)

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 number of ticks per capture can be adjusted to give a lower average frame time.

See the *`set_ticks_per_capture()`* method.

3.3.2 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.

3.3.3 Change Render Quality

You can adjust Holodeck 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

3.4 Using Holodeck Headless

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

3.4.1 Headless Mode vs Disabling Viewport Rendering

These are two different features.

Disabling Viewport Rendering is calling the (*should_render_viewport()*) method on a *HolodeckEnvironment*. 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.

3.4.2 Forcing Headless Mode

In holodeck.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

---

3.4.3 Automatic Headless Mode

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

Note: This will not disable viewport rendering.

3.5 Configuring Weather and Time

Holodeck worlds have weather and time that can be configured, either with *Scenarios* or programmatically in real time.

See the *`WeatherController`* documentation for reference.

3.5.1 Weather Options

Type

Holodeck worlds have three possible types of weather: `sunny` `cloudy` and `rain`.

In a scenario

```
{
   "weather": {
      "type": "rain"
   }
}
```

Programmatically

```
env = holodeck.make("...")
env.weather.set_weather("rain")
```

Fog depth

Fog depth is set on scale from `0` to `1`.

In a scenario

```
{
   "weather": {
      "fog_depth": 0.5
   }
}
```

Programmatically

```
env = holodeck.make("...")
env.weather.set_fog_density(0.5)
```

Day length

Note: By default, the day cycle in Holodeck worlds is turned off and time is fixed.

The day cycle length is set in minutes.

In a scenario

```
{
    "weather": {
        "day_cycle_length": 60
    }
}
```

Programmatically

```
env = holodeck.make("...")
env.weather.start_day_cycle(60)
```

Time

The time of the day can be set as a number between `0` and `23` inclusive.

In a scenario

```
{
    "weather": {
        "hour": 12
    }
}
```

Programmatically

```
env = holodeck.make("...")
env.weather.set_day_time(12)
```

CHAPTER

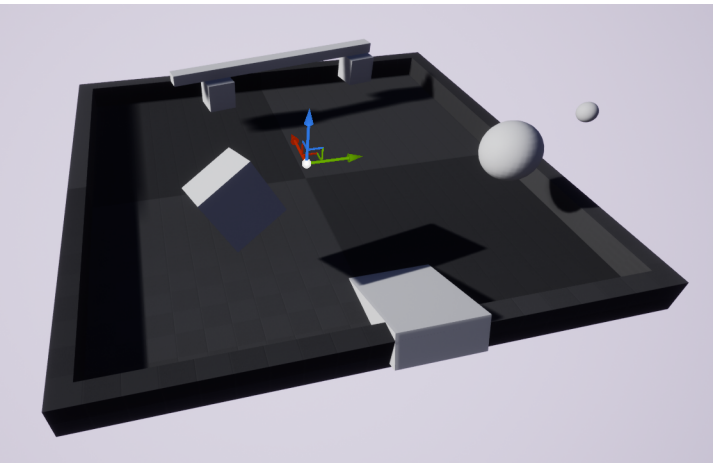
FOUR

HOLODECK PACKAGES

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

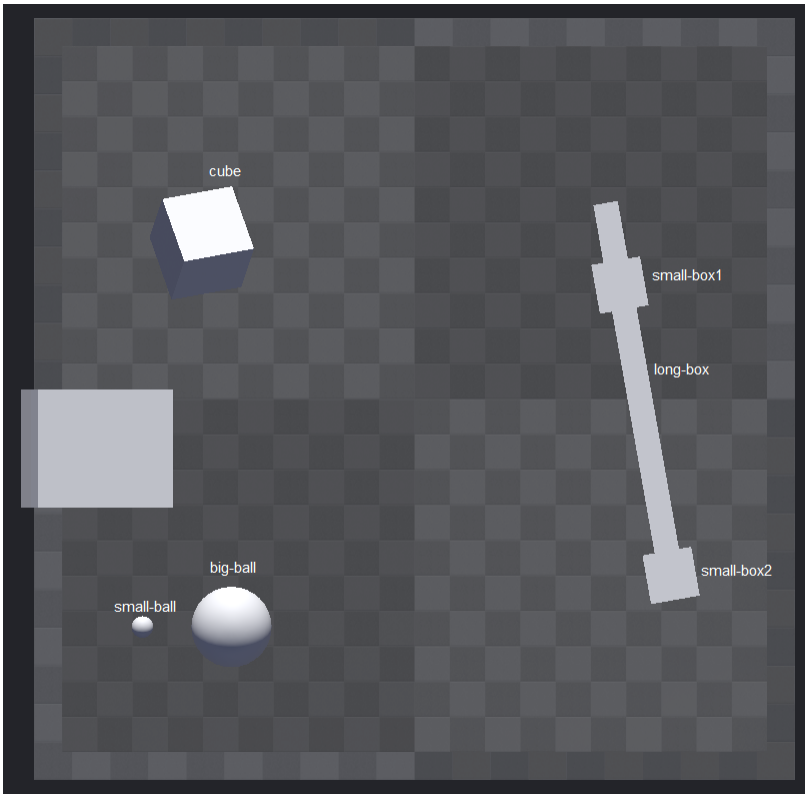
4.1 DefaultWorlds Package

4.1.1 AndroidPlayground



This is a small world with some props agents can interact with. It was designed for the android to interact with.

Layout



Tagged Items

- `cube`
- `small-ball`
- `big-ball`
- `small-box1`
- `small-box2`
- `large-box`

AndroidPlayground-MaxDistance

Type: *Distance Task*

This scenario rewards the agent for maximizing its distance from the start location.

Agents

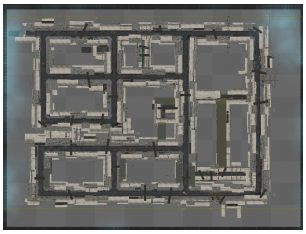
- `android0`: Main *Android* agent

See AndroidPlayground-MaxDistance.json.

4.1.2 CyberPunkCity

This world is a dark, moody environment. It has a humanoid figure running around, the intention is that agents learn to follow it.

Layout



CyberPunkCity-Follow.rst

Type: *Follow Task*

This scenario rewards the UAV for following the NavAgent and keeping it in sight.

Agents

- `uav0`: *UAV*, Main agent,
- `nav0`: *NavAgent* that will automatically navigate to a predetermined location

See CyberPunkCity-FollowSight.json.

4.1.3 EuropeanForest

This world is a very large and contains a few different environments. It has plains, forest, and a few structures for agents to interact with.

Layout

EuropeanForest-MaxDistance

Type: *Distance Task*

This scenario rewards the agent for maximizing its distance from the start location.

Agents

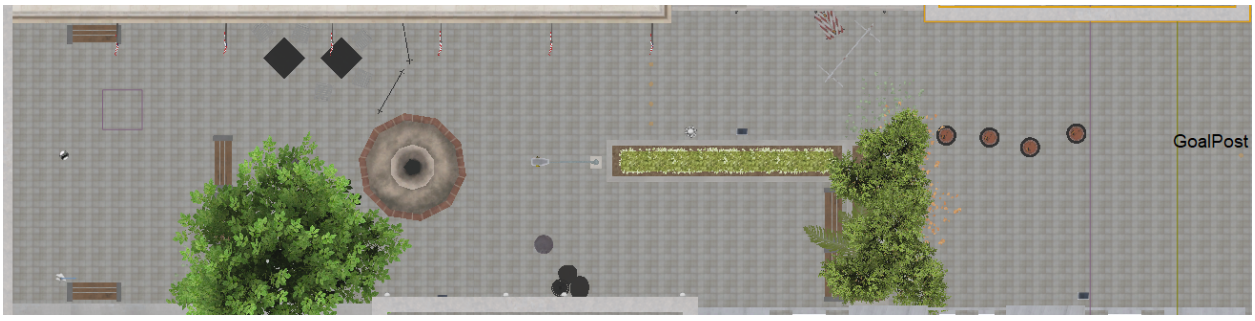
- `uav0`: Main *UAV* agent

See EuropeanForest-MaxDistance.json.

4.1.4 MazeWorld

This is a small, linear world meant to help agents learn to navigate around obstacles.

Layout



Tagged Items

- `GoalPost`

MazeWorld-FinishMazeSphere

The goal of this task is for the sphere agent to finish the maze and get as close as possible to post with a golden ball on the other end.

Agents

- `sphere0`: Main *sphere* agent

See MazeWorld-FinishMazeSphere.json.

4.1.5 InfiniteForest

This world is a randomly generated forest that will generate wherever the main agent goes. Each time the world is initialized, it will have a different layout.

InfiniteForest-MaxDistance

Type: *Distance Task*

This scenario rewards the agent for maximizing its distance from the start location.

Agents

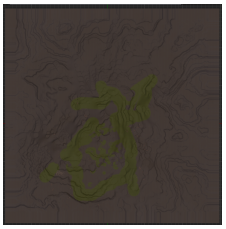
- `uav0`: Main *UAV* agent

See InfiniteForest-MaxDistance.json.

4.1.6 RedwoodForest

This is a forest setting with more diverse tree sizes. It has a fixed size.

Layout



RedwoodForest-MaxDistance

Type: *Distance Task*

This scenario rewards the agent for maximizing its distance from the start location.

Agents

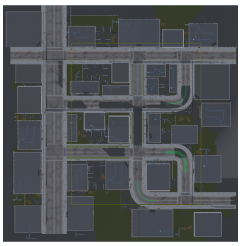
- `uav0`: Main *UAV* agent

See RedwoodForest-MaxDistance.json.

4.1.7 Urban City

Urban City contains a few city blocks and roads, meant to help agents learn to navigate an urban setting.

Layout



UrbanCity-MaxDistance

Type: *Distance Task*

This scenario rewards the agent for maximizing its distance from the start location.

Agents

- `uav0`: Main *UAV* agent

See UrbanCity-MaxDistance.json.

4.2 Dexterity package

4.2.1 Playroom

This is a small playroom with movable toys.

Layout

Playroom-Android

This scenario allows the *AndroidAgent* to interact with the playroom. There is no reward or objective, as the goal of this task is to encourage play.

Agents

- `android0`: Main *Android* agent

See Playroom-Android.json .

Playroom-Hand

This scenario allows the *HandAgent* to interact with the playroom. There is no reward or objective, as the goal of this task is to encourage play.

Agents

- `hand0`: Main *Hand* agent

See Playroom-Hand.json .

Playroom-StandFromGround

Type: *Location Task*

In this scenario the *AndroidAgent* must stand up, as measured by its head position. The android begins laying on the ground.

Agents

- `android0`: Main *AndroidAgent*

See Playroom-StandFromGround.json.

Playroom-StandFromStanding

Type: *Location Task*

In this scenario the *AndroidAgent* must stand itself back up from the standing position, as measured by the android's head position.

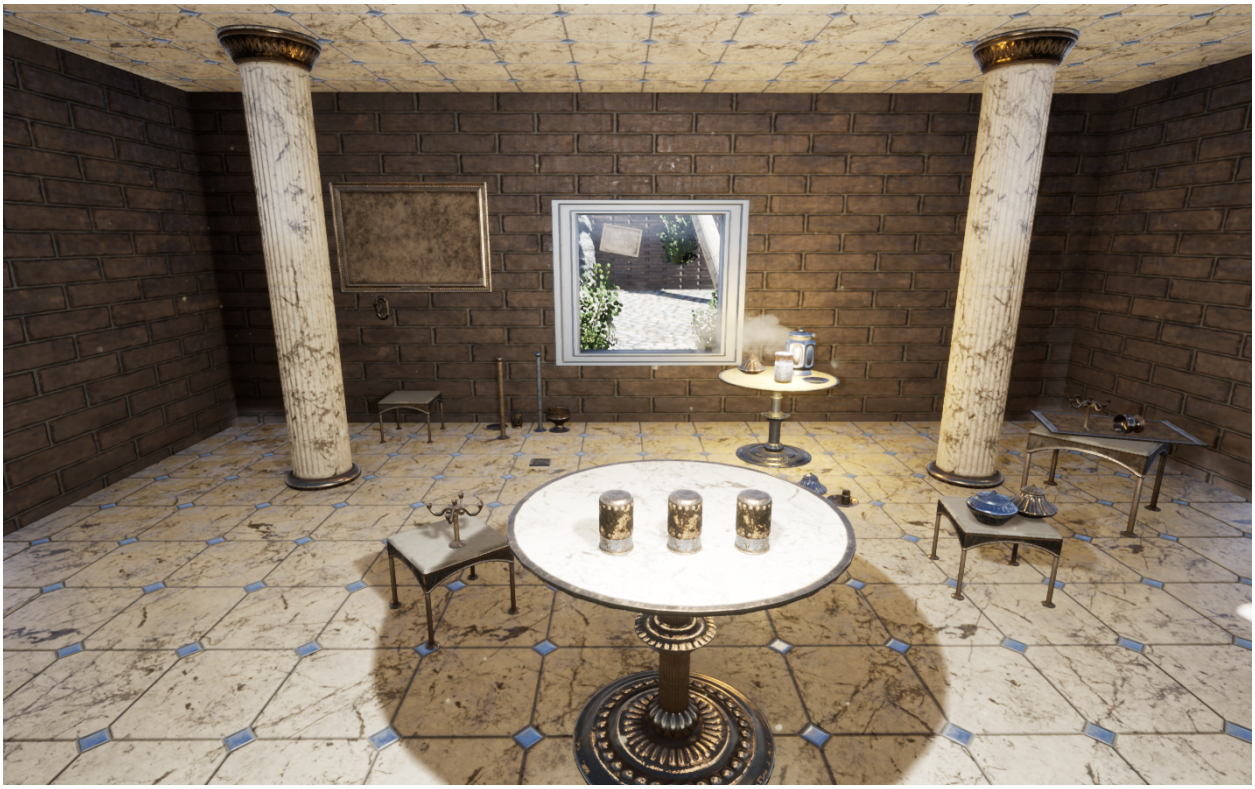
Since the android is initially standing, it will initially receive some reward.

Agents

- `android0`: Main *AndroidAgent*

See Playroom-StandFromStanding.json.

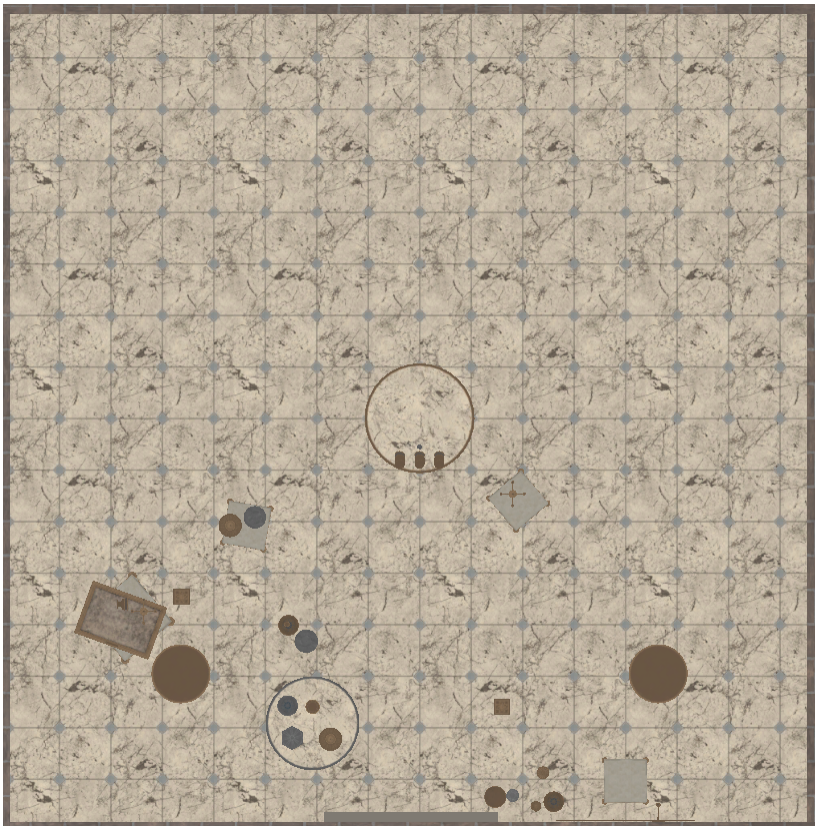
4.2.2 CupGame



This is a small room with a game of ball and cups sitting on a table. In order to activate the game, an agent must have the CupGameTask added to it.

If you want to reconfigure the task (change the number of shuffles, change the speed, etc), call the `start_game()` method on the `CupGameTask` class or alter the config file (see the *configuration* of the *Cup Game Task*.)

Layout



CupGame-Custom

Type: *Cup Game Task*

This scenario has a hand agent positioned directly in front of the cup game. The game is not automatically set up with any configuration, which requires the user to call the `start_game()` method and manually configure the game.

Agents

- `hand0`: Main *Hand* agent

See CupGame-Custom.json.

CupGame-Easy

Type: *Cup Game Task*

This scenario has a hand agent positioned directly in front of the cup game. The game is configured to be fairly easy with only `3` shuffles and the lowest speed multiplier of `1`.

Agents

- `hand0`: Main *Hand* agent

See CupGame-Easy.json.

CupGame-Hard

Type: *Cup Game Task*

This scenario has a hand agent positioned directly in front of the cup game. The game is configured to be more difficult with `10` shuffles and a higher speed multiplier of `3`.

Agents

- `hand0`: Main *Hand* agent

See CupGame-Hard.json.

4.2.3 Clean Up

This is an alleyway with a trash can in the middle. A *Clean Up Task* task can be used to spawn trash around the can and give a reward based on the amount of trash collected in the can.

Layout

CleanUp-GroundAndroid

Type: *Clean Up Task*

This scenario gives an android a reward based on the number of pieces of trash it picks up off the ground and puts in the trash can. The trash is spawned on the ground around the trash can.

Agents

- `android0`: Main *Android* agent

See CleanUp-GroundAndroid.json.

CleanUp-GroundHand

Type: *Clean Up Task*

This scenario gives a hand agent a reward based on the number of pieces of trash it picks up off the ground and puts in the trash can. The trash is spawned on the ground around the trash can.

Agents

- `hand0`: Main *Hand* agent

See CleanUp-GroundHand.json.

CleanUp-TableAndroid

Type: *Clean Up Task*

This scenario gives an android a reward based on the number of pieces of trash it pushes from a table to the adjacent the trash can. This is intended to be an easier version of other cleanup tasks, since the agent does not have to pick up the trash, it can simply slide it off the table.

Agents

- `android0`: Main *Android* agent

See CleanUp-GroundAndroid.json.

CleanUp-TableHand

Type: *Clean Up Task*

This scenario gives a hand agent a reward based on the number of pieces of trash it pushes from a table to the adjacent the trash can. This is intended to be an easier version of other cleanup tasks, since the agent does not have to pick up the trash, it can simply slide it off the table.

Agents

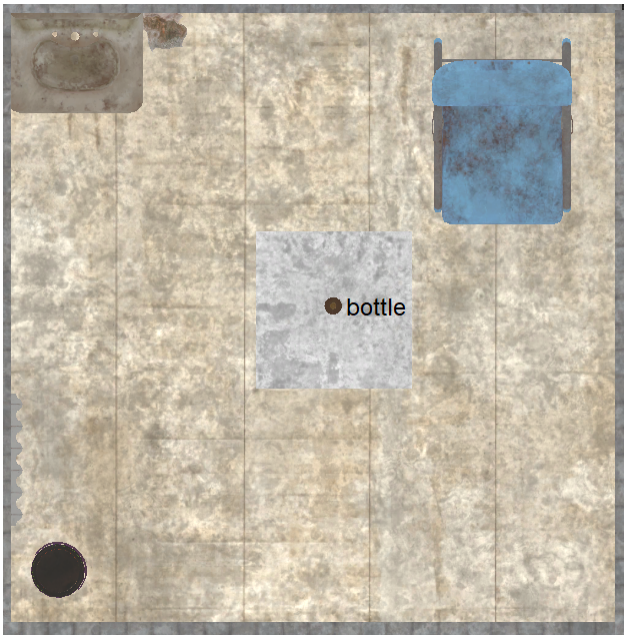
- `hand0`: Main *Hand* agent

See CleanUp-TableHand.json.

4.2.4 Grip

This is a very small room with a bottle sitting on a platform in the middle. The bottle has simulated physics.

Layout



Tagged Items

- `bottle`

Grip-LiftBottle

Type: *Location Task*

In this scenario the hand agent must lift the bottle to a certain point. The scenario is intended to teach the agent motor control and basic object manipulation.

Agents

- `hand0`: Main *Hand* agent

See Grip-LiftBottle.json .

4.2.5 AndroidPlaytime

This is a small playroom with small, movable toys and windows.

Layout

AndroidPlaytime-PlayRoom

This scenario has an android agent spawn in the AndroidPlaytime room. There is no reward or objective, the goal of this task is to encourage play.

Agents

- `android0`: Main *Android* agent

4.3 Package Structure

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

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

4.3.1 Package Contents

The `.zip` file must contain the following elements

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

4.3.2 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
```

4.3.3 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,
      }
   ]
}
```

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

4.4 Scenarios

4.4.1 What is a scenario?

A scenario tells Holodeck 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
- Tasks
  - Which task
  - Which agents play which role in the task

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 `holodeck.make()` to create an environment, you pass in the name of a scenario, eg `holodeck.make("UrbanCity-Follow")`. This tells Holodeck which world to load and where to place agents.

4.4.2 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}",
    "agents":[
        "array of agent objects"
    ],
    "weather": {
        "hour": 12,
        "type": "'sunny' or 'cloudy' or 'rain'",
        "fog_density": 0,
```

(continues on next page)

(continued from previous page)

```
        "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 *Configuring Weather and Time*.

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

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: Holodeck coordinates are **left 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.

The location value is measured in meters, in the format `[dx, dy, dz]` and the rotation is `[roll, pitch, yaw]`.

Agent Types

Here are valid `agent_type` s:

Agent Type	String in agent_type
AndroidAgent	`AndroidAgent`
HandAgent	`HandAgent`
TurtleAgent	`TurtleAgent`
NavAgent	`NavAgent`
SphereAgent	`SphereAgent`
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 \"\"",
    "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 and a location/rotation**

   The sensor will be placed offset to the socket by the location and rotation

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

3. **Provide just a location/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]
   }
   ```

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": "",
    "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
    }
}
```

4.5 Tasks

Below are the different tasks available for use in Holodeck.

4.5.1 Distance Task

The distance tasks calculates a dense distance based reward. The agent will receive a reward of 1 as it crosses intervals that are a certain distance away from a goal location. It can be configured to have the agent either maximize or minimize the distance from a location, actor or the agent's starting location.

Configuration

Each of the following parameters can be placed in the configuration field for a distance task sensor (see *scenario files*.)

DistanceActor

The reward is calculated by measuring the distance between the distance actor and the goal actor/location. If left empty, it will default to the task's agent.

- `"DistanceActor": "name-of-actor"`

Goal

The distance between the goal actor/location and the distance actor is used to calculate the reward. Only the GoalActor or GoalLocation can be set, not both.

`"GoalActor": "name-of-actor"`

or

`"GoalLocation": [1.0, 2.0, 3.0]`

Interval

The interval controls the distance an agent must cover before it receives a reward.

`"Interval": 5`

GoalDistance

This distance is used to determine if the task has reached its terminal state and the agent has travelled far enough away.

`"GoalDistance": 1`

MaximizeDistance

Boolean value to indicate if the distance should be maximized or minimized. If left empty, it defaults to false.

`"MaximizeDistance": true`

3dDistance

Boolean value to indicate whether to incorporate height into the distance calculation. If false, it will only use the xy values and ignore vertical distance. If left empty, it defaults to false.

`"3dDistance": true`

Example

```
{
    "DistanceActor": "baseball",
    "GoalActor": "target",
    "Interval": 5,
    "GoalDistance": 0.2,
    "MaximizeDistance": false
}
```

4.5.2 Location Task

Calculates a sparse distance reward based on the distance to a location or an actor. Can maximize or minimize a distance. A reward of 1 is only given only if the agent reaches the goal target within the goal distance.

Configuration

Each of the following parameters can be placed in the configuration field for a location task sensor (see *scenario files*.)

LocationActor

The reward is given based on the distance between this actor and the goal target. Defaults to the task's agent.

`"LocationActor": "name-of-actor"`

GoalTarget

The Location task needs an actor or a location to use in the distance calculations.

`"GoalActor": "name-of-actor"`

or

`"GoalLocation": [3.14, 2.71, 117]`

GoalDistance

This is the distance from the goal target the `LocationActor` must be to get a reward and terminal.

`"GoalDistance": 1024.0`

NegativeReward

A boolean representing whether reaching the goal target returns `1` or `-1`. Defaults to false.

`"NegativeReward": false`

HasTerminal

A boolean representing whether reaching the goal target returns a terminal value or not. Defaults to false.

`"HasTerminal": true`

Example

```
{
    "LocationActor": "golf-ball",
    "GoalActor": "cup",
    "GoalDistance": 1024.0,
    "NegativeReward": false,
    "HasTerminal": true
}
```

4.5.3 Follow Task

The follow task calculates a reward based on the distance to an actor and optionally if the agent has line of sight to it.

If OnlyWithinSight is true, the reward is set to the percent distance covered from the MinDistance to the ToFollow target *if* the angle from the agent to target is less than FOVRadians *and* is there is nothing blocking the agent's line of sight. Otherwise the reward is 0.

If OnlyWithinSight is false, then the reward is set to the the percent distance covered from the MinDistance to the ToFollow Actor.

The reward will be a value 0 to 100

Configuration

Each of the following parameters can be placed in the configuration field for a follow task sensor (see *scenario files*.)

ToFollow

Name of the actor to follow.

```
"ToFollow": "name-of-actor"
```

OnlyWithinSight

Boolean value indicating if the reward should be calculated only when the actor to follow is within the agent's field of view.

```
"OnlyWithinSight": true
```

FOVRadians

Float value, the field of view of the agent, in radians. See above how this is used in the reward calculation.

```
"FOVRadians": 1.5
```

MinDistance

Float value, used to specify the minimum distance

```
"MinDistance": 512.0
```

FollowSocket

The socket of the ToFollow actor the agent needs to see if OnlyWithinSight is true. If left empty, it defaults to the actor's location.

```
"FollowSocket": "head"
```

Example

```
{
    "ToFollow": "person",
    "OnlyWithinSight": true,
    "FOVRadains": 2.0,
    "MinDistance": 1024.0,
    "FollowSocket": "head"
}
```

4.5.4 Avoid Task

The avoid task calculates a reward based on the distance between the agent and an actor to avoid, with an option to incorporate whether the actor to avoid can see the agent.

If `OnlyWithinSight` is `false`, then the reward is set to the the percent distance covered from the MinDistance to the ToAvoid Actor. The closer ToAvoid is, the lower the reward.

If `OnlyWithinSight` is `true`, the reward calculation is the same as above *if* the angle from the ToAvoid to the agent is less than FOVRadians *and* is there is nothing blocking the ToAvoid's line of sight. Otherwise the reward is 100.

The reward will be a value 0 to 100

Configuration

Each of the following parameters can be placed in the configuration field for an avoid task sensor (see *scenario files*.)

ToAvoid

Name of the actor to avoid.

`"ToAvoid": "name-of-actor"`

OnlyWithinSight

Boolean value indicating if the reward should be calculated only when the agent is within the ToAvoid actor's field of view.

`"OnlyWithinSight": true`

FOVRadians

Float value, the field of view of the agent, in radians. See above how this is used in the reward calculation.

`"FOVRadians": 1.5`

MinDistance

Float value, used to specify the minimum distance.

`"MinDistance": 512.0`

StartSocket

The socket of the ToAvoid actor that its vision is calculated from.

`"StartSocket": "head"`

EndSocket

The socket of the agent that the ToAvoid actor needs to see for the vision calculation.

`"EndSocket": "body"`

Example

```
{
    "ToAvoid": "hunter",
    "OnlyWithinSight": true,
    "FOVRadains": 2.0,
    "MinDistance": 1000.0,
    "StartSocket": "head",
    "EndSocket": "body"
}
```

4.5.5 Cup Game Task

Calculates reward based on whether the correct cup is touched and whether the ball is touched.

- A reward of `1` is given for one tick if the correct cup is touched and no other cups are touched.
- A reward of `2` and terminal is given when the ball itself is touched and no incorrect cups are touched.
- A reward of `-1` and terminal is given when an incorrect cup is touched.

The cup game task only works in the CupGame world in the dexterity package. The game will not start if a cup game task is not added to an agent.

Configuration

Each of the following parameters can be placed in the configuration field for a cup game task sensor (see *scenario files*.)

The configuration can also be set programmatically by calling *`start_game()`* if the sensor has no configuration block. That configuration will reset after every call to `reset()`.

Speed Multiplier

`1.0` is the base speed, cups will rotate faster with a higher multiplier. It is best to keep values between `1` and `10`.

`"Speed": 2.4`

NumShuffles

Number of times the cups are exchanged.

`"NumShuffles": 10`

Seed

Seed for the RNG used to shuffle the cups. Providing a fixed seed will result in a deterministic set of exchanges, which may be useful for training.

If left empty or not defined, the cups will rotate using a randomly generated seed.

`"Seed": 1`

Example

```
{
    "Speed": 5.0,
    "NumShuffles": 1,
    "Seed": 0
}
```

4.5.6 Clean Up Task

Initializes the clean up task in the world. This task only works in the CleanUp world in the Dexterity package where there is a trash can in the middle of the map.

Trash will spawn randomly around the trash can when the task starts. A clean up task must be added to an agent for the task to start.

The reward is based on the number of pieces of trash placed in the trash can. For each piece of trash added to the can, a reward of `1` is given. For each piece of trash removed, a reward of `-1` is given. If all the trash is in the can, terminal is given.

If `UseTable` is `true` a table will spawn next to the trash can, all trash will be on the table, and the trash can lid will be absent. This makes the task significantly easier. If `false`, all trash will spawn on the ground.

Configuration

Each of the following parameters can be placed in the configuration field for a clean up task sensor (see *scenario files*.)

The configuration can also be set programmatically by calling `start_task()`. Do not call if the config file has a configuration block. That configuration will reset after every call to `reset()`.

NumTrash

Int representing the amount of trash to spawn around the trash can.

`"NumTrash": 6`

UseTable

Boolean value representing whether to use the simpler table task configuration.

`"UseTable": false`

Example

```
{
    "NumTrash": 5,
    "UseTable": false
}
```

These tasks can be configured and used in *scenario files*.

4.6 Package Installation Location

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

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

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

This is the path returned by `holodeck.util.get_holodeck_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*.

4.6.1 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

HOLODECK AGENTS

Documentation on specific agents available in Holodeck:

5.1 AndroidAgent

5.1.1 Images

5.1.2 Description

An android agent that can be controlled via torques supplied to its joints. See `AndroidAgent` for more details.

5.1.3 Control Schemes

Android Direct Torques (`0`) A 94 dimensional vector of continuous values representing torques to be applied at each joint. See *Android Joints* below for a description of the joint indicies.

Android Max Scaled Torques (`1`) A 94 dimensional vector of continuous values between [-1, 1] representing the scaled torques to be applied at each joint. See *Android Joints* below for a description of the joint indicies.

1 represents the maximum forward torque and -1 the maximum torque in the opposite direction.

5.1.4 Android Joints

The control scheme for the android and the *JointRotationSensor* use a 94 length vector refer to 48 joints.

To gain insight into these joints, refer to the table below, or use the *joint_ind()* helper method to convert a name (eg `spine_02`) to and index (`6`).

Note: Note that the index given is the start index for the joint, see the section header for how many values after this index each joint has.

Example: `neck_01` starts at index 3, and has `[swing1, swing2, twist]` , so index 3 in the 94 length vector corresponds to `swing1`, 4 corresponds to `swing2`, and 5 corresponds to `twist` for `neck_01`.

Returned in the following order:

Head, Spine, and Arm joints Each has `[swing1, swing2, twist]`	
0	head
3	neck_01
6	spine_02
9	spine_01
12	upperarm_l
15	lowerarm_l
18	hand_l
21	upperarm_r
24	lowerarm_r
27	hand_r
Leg Joints Each has `[swing1, swing2, twist]`	
30	thigh_l
33	calf_l
36	foot_l
39	ball_l
42	thigh_r
45	calf_r
48	foot_r
51	ball_r
First joint of each finger Has only `[swing1, swing2]`	
54	thumb_01_l
56	index_01_l
58	middle_01_l
60	ring_01_l
62	pinky_01_l
64	thumb_01_r
66	index_01_r
68	middle_01_r
70	ring_01_r
72	pinky_01_r
Second joint of each finger Has only `[swing1]`	

continues on next page

Table 1 – continued from previous page

74	thumb_02_l
75	index_02_l
76	middle_02_l
77	ring_02_l
78	pinky_02_l
79	thumb_02_r
80	index_02_r
81	middle_02_r
82	ring_02_r
83	pinky_02_r
Third joint of each finger Has only [swing1]	
84	thumb_03_l
85	index_03_l
86	middle_03_l
87	ring_03_l
88	pinky_03_l
89	thumb_03_r
90	index_03_r
91	middle_03_r
92	ring_03_r
93	pinky_03_r

5.1.5 AndroidAgent Bones

The *RelativeSkeletalPositionSensor* returns an array with four entries for each bone listed below.

Index	Bone Name
0	pelvis
4	spine_01
8	spine_02
12	spine_03
16	clavicle_l
20	upperarm_l
24	lowerarm_l
28	hand_l
32	index_01_l
36	index_02_l
40	index_03_l
44	middle_01_l
48	middle_02_l
52	middle_03_l
56	pinky_01_l
60	pinky_02_l
64	pinky_03_l
68	ring_01_l
72	ring_02_l
76	ring_03_l
80	thumb_01_l

continues on next page

Table 2 – continued from previous page

Index	Bone Name
84	thumb_02_l
88	thumb_03_l
92	lowerarm_twist_01_l
96	upperarm_twist_01_l
100	clavicle_r
104	upperarm_r
108	lowerarm_r
112	hand_r
116	index_01_r
120	index_02_r
124	index_03_r
128	middle_01_r
132	middle_02_r
136	middle_03_r
140	pinky_01_r
144	pinky_02_r
148	pinky_03_r
152	ring_01_r
156	ring_02_r
160	ring_03_r
164	thumb_01_r
168	thumb_02_r
172	thumb_03_r
176	lowerarm_twist_01_r
180	upperarm_twist_01_r
184	neck_01
188	head
192	thigh_l
196	calf_l
200	calf_twist_01_l
204	foot_l
208	ball_l
212	thigh_twist_01_l
216	thigh_r
220	calf_r
224	calf_twist_01_r
228	foot_r
232	ball_r
236	thigh_twist_01_r

5.1.6 Sockets

- `CameraSocket` located in the middle of the android's face
- `Viewport` located behind the agent
- All of the joints may be used as sockets. See *Android Joints*.

5.2 HandAgent

5.2.1 Images

5.2.2 Description

A floating hand agent that can be controlled by applying torques to joints and moved around in three dimensions.

5.2.3 Control Schemes

- **Raw Joint Torques** (`0`)

  23 length vector of raw torques to pass into the joints, in the order listed beow in *HandAgent Joints*

- **Scaled Joint Torques** (`1`)

  23 length vector of scaled torques, between `-1` and `1`. The strength finger each joint is scaled depending on the weight of the bone and if it is a finger or not. `1` represents the maximum power in the forward direction

- **Scaled Joint Torques + Floating** (`2`)

  Same as above, but the vector is of length 26, with the last three values representing the amount of movement in the `[x, y, z]` directions (see *Coordinate System*), with a maximum of `0.5` meters of freedom.

  The last coordinates allow the HandAgent to float around.

5.2.4 HandAgent Joints

The control scheme for the HandAgent and the *JointRotationSensor* use a 94 length vector refer to 48 joints.

To gain insight into these joints, refer to the table below.

Note: Note that the index given is the start index for the joint, see the section header for how many values after this index each joint has.

Example: `hand_r` starts at index 0, and has `[swing1, swing2, twist]`, so index 0 in the vector corresponds to `swing1`, 1 corresponds to `swing2`, and 2 corresponds to `twist` for `hand_r`

Returned in the following order:

Arm joints Each has `[swing1, swing2, twist]`	
0	hand_r
First joint of each finger Has only `[swing1, swing2]`	
64	thumb_01_r
66	index_01_r
68	middle_01_r
70	ring_01_r
72	pinky_01_r
Second joint of each finger Has only `[swing1]`	
79	thumb_02_r
80	index_02_r
81	middle_02_r
82	ring_02_r
83	pinky_02_r
Third joint of each finger Has only `[swing1]`	
89	thumb_03_r
90	index_03_r
91	middle_03_r
92	ring_03_r
93	pinky_03_r

5.2.5 HandAgent Bones

The *RelativeSkeletalPositionSensor* returns an array with four entries for each of the 17 bones listed below.

Index	Bone Name
0	lowerarm_r
4	hand_r
8	index_01_r
12	index_02_r
16	index_03_r
20	middle_01_r
24	middle_02_r
28	middle_03_r
32	pinky_01_r
36	pinky_02_r
40	pinky_03_r
44	ring_01_r
48	ring_02_r
52	ring_03_r
56	thumb_01_r
60	thumb_02_r
64	thumb_03_r

5.2.6 Sockets

- CameraSocket located behind and above the wrist
- Viewport located looking at the agent from the side
- All of the joints may be used as sockets. See *HandAgent Joints*

5.3 NavAgent

5.3.1 Images

See *holodeck.agents.NavAgent* for more details.

5.3.2 Description

The NavAgent is not meant for training, it is meant to be used as an objective in tasks. Given world coordinates, it will use use Unreal's AI system to attempt to intelligently navigate towards those coordinates.

5.3.3 Control Schemes

Nav Target Location (``0``) A 3-length floating point vector used to specify the x, y and z coordinates for the agent to navigate to.

5.3.4 Sockets

None.

5.4 SphereAgent

5.4.1 Images

5.4.2 Description

A basic sphere robot that moves along a plane. The SphereAgent does not have physics - it simply computes its next location and teleports there, as compared to the *TurtleAgent* which has mass and momentum.

See `SphereAgent` for more details.

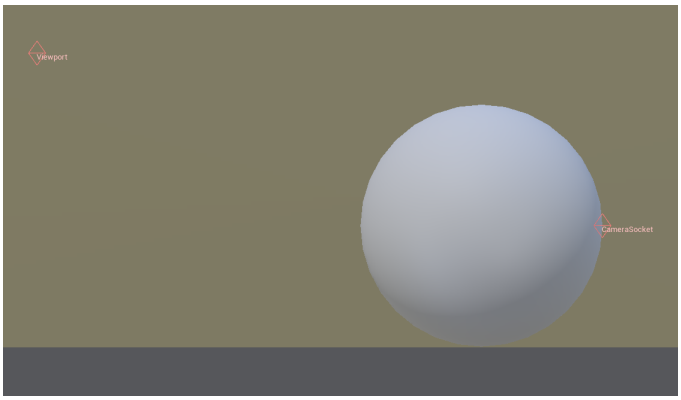
5.4.3 Control Schemes

Sphere discrete (`0`) A single-length integer vector that accepts 1 of four possible numbers; 0: move forward, 1: move backward, 2: turn right, 3: turn left

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

5.4.4 Sockets

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



5.5 TurtleAgent

5.5.1 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 *SphereAgent* which teleports around. The TurtleAgent is subject to gravity and can climb ramps and slopes.

See `TurtleAgent` for more details.

5.5.2 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).

5.5.3 Sockets

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

5.6 UavAgent

5.6.1 Images

5.6.2 Description

A quadcopter UAV agent.

See the *UavAgent* class.

5.6.3 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.

5.6.4 Sockets

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

CHAPTER

SIX

CHANGELOG

6.1 Holodeck 0.3.1

04/02/2020

More bug fixes, improvements, and even a few new features.

6.1.1 Highlights

- **Holodeck now requires Python 3.5 or greater**
- Added `AbuseSensor` and `RangeFinderSensor`
- Added programmatic spawning of props, see `spawn_prop()`
- Weather can be specified in scenarios, see *Configuring Weather and Time*.

6.1.2 New Features

- Added optional start location and rotation randomization on `reset()`. See *Location Randomization*. (#295)
- `spawn_prop()` now allows basic objects (spheres, cubes, cylinders) to be spawned at arbitrary locations in the environment. (#397)
- *Distance Task* by default now calculates the distance to the objective along the XY plane, to discourage flying straight up. (#360)

  If the full 3D distance is desired, set the `3dDistance` flag in the configuration block of the *Distance Task*. (#360)
- Added `AbuseSensor`, which senses if an agent has been abused. Agents experience abuse when they fall from a high distance or other agent-specific situations. (#262)
- Environment weather/time can be optionally configured with *Scenarios* (#263). See *Configuring Weather and Time*.
- `set_weather()` now has sunny weather available, which allows you to revert back to the default weather. (#376)
- Added `RangeFinderSensor` which calculates the distance from the sensor to the first collision in the environment. The sensor can send out multiple rays in a circle if desired.

6.1.3 Changes

- **Holodeck now requires Python >= 3.5** (#389)
- Moved weather/time methods from *HolodeckEnvironment* to new *WeatherController* (#196, #263)
- Calling *send_world_command()* for an environment without the given command will now cause the environment to exit rather than fail silently. This includes all relevant methods in the *WeatherController*.
- Removed the ability to toggle sensors during runtime with the removal of `SetSensorEnabledCommand`, `set_sensor_enabled()`, and `set_sensor_enable()`. To specify which sensors to include, use *Custom Scenario Configurations*. (#268)
- Improved Docker images. See *Docker Installation*. (#347)
  - Tests can now be run inside of Docker containers
  - All images are based on Ubuntu 18.04 now
  - Added image for Dexterity package, and an image with every package
- Every control scheme now has limits on inputs (ie maximum or minimum thrust) (#369)

  See *get_high()* and *get_low()* to read them.
- Scenario Changes:
  - **EuropeanForest-MaxDistance, RedwoodForest-MaxDistance, UrbanCity-MaxDistance:** Added *AbuseSensor*
  - **InfiniteForest-MaxDistance:** Added *AbuseSensor* and *RangeFinderSensor*.
  - **MazeWorld-FinishMazeSphere:** Added *RangeFinderSensor*

6.1.4 Bug Fixes

- Fixed UAV blades rotating incorrectly (thanks @sethmnielsen!) (#331)
- Fixed some `posix_ipc.BusyError: Semaphore is busy` errors on Linux systems when creating a scenario (#285)
- Fixed a bug where the UE4 editor crashes when an agent is manually added to a level (#361)
- Fixed crash when manually disabling viewport when it would've been disabled anyway. (#378)
- Fixed SphereAgent having the incorrect default control scheme (#350)

6.2 Holodeck 0.3.0

11/02/2019

This is a content release focused on improving the *AndroidAgent* and adding more scenarios and tasks for it. We also added a new floating hand agent to provide a simpler agent that can do many of the dexterity tasks.

6.2.1 Highlights

- Added dexterity-package with new worlds and scenarios (see below for comprehensive listing)
- Added *Clean Up Task* and *Cup Game Task* tasks
- Added *HandAgent*

6.2.2 New Features

- Added the dexterity-package with new worlds and scenarios:
  - *Playroom*
    - *Playroom-Android*
    - *Playroom-Hand*
    - *Playroom-StandFromGround*
    - *Playroom-StandFromStanding*
  - *Clean Up* (#290)
    - *CleanUp-GroundAndroid*
    - *CleanUp-GroundHand*
    - *CleanUp-TableAndroid*
    - *CleanUp-TableHand*
  - *CupGame* (#288)
    - *CupGame-Custom*
    - *CupGame-Easy*
    - *CupGame-Hard*
  - *Grip*
    - *Grip-LiftBottle*
- Added the *HandAgent* - a simplified Android hand that can float around (#287)
  - HandAgent can be used with the same Android-specific sensors (`JointRotationSensor`, `PressureSensor`, `RelativeSkeletalPositionSensor`)
- Added new tasks sensors for specific worlds
  - *Cup Game Task* (#318)
  - *Clean Up Task* (#321)
- Packages can be installed directly from a URL (see `install`) (#129)
- Agent sensors can now be rotated at run time with `rotate()`. (#305)
- The config files can now specify whether an agent should be spawned (#303)
- Pressing `h` now shows the coordinates of the agent the viewport is following or the coordinates of the camera if it is detached (see *Hotkeys*). (#253)
- The viewport now follows the main agent as specified in the config file by default. (#238)
- You can now specify the number of ticks you want to occur in the `tick()` and the `step()` methods, (#313)

6.2.3 Changes

- Increased the *AndroidAgent*'s strength in the `ANDROID_MAX_SCALED_TORQUES` control scheme.
  - Previously the AndroidAgent didn't have enough strength to even move its legs.
  - Strength was approximately doubled (See JointMaxTorqueControlScheme.h)
- Location sensor now returns the location of the sensor, not just the agent (#306)
- Updated to Unreal Engine 4.22 (#241)
- *TurtleAgent* is now subject to gravity, has increased power, is black, and slightly smaller. (#217)
- Removed the `set_state()` and `teleport()` methods from the *HolodeckEnvironment* class.

  These methods were duplicates of the corresponding methods on the *HolodeckAgent* class. See the linked issue for migration suggestions (#311)
- Removed the `get/set_ticks_per_capture` methods from the *HolodeckAgent* and *HolodeckEnvironment* classes, moved *set_ticks_per_capture()* method to the *RGBCamera* class. (#197)
- Viewport will now follow the main agent by default. (#238)
- Viewport will not be rendered when it is hidden (`show_viewport` param in *HolodeckEnvironment*, Linux only) (#283)

6.2.4 Bug Fixes

- Fixed the *RelativeSkeletalPositionSensor*.
  - This sensor returns the location of bones, not sensors. Since there are more bones than joints, previously it returned them in a completely different order than expected.
  - Now the order for this sensor is explicitly specified in *AndroidAgent Bones* and *HandAgent Bones*.
  - Previously on the first tick it would return uninitialized garbage on the first tick
- Fixed being unable to spawn the *TurtleAgent*. (#308)
- Fixed the *set_physics_state()* method. (#311)
- Fixed agent spawn rotations being in the incorrect order. Fixed the documentation that specified the incorrect order as well (*Rotations*) (#309)
- Fixed being unable to set the ticks per capture of a camera if it was not named `RGBCamera`. (#197)
- Fixed being unable to make a Holodeck window larger than the current screen resolution (#301)
- Fixed being unable to configure *ViewportCapture* sensor. (#301)

6.2.5 Known Issues

- The TurtleAgent does not move consistently between Linux and Windows. (#336)

6.3 Holodeck 0.2.2

06/20/2019

This is mostly a maintenance release focused on cleaning up bugs that were unresolved in `0.2.1`

6.3.1 New Features

- When freecamming around, *pressing shift* moves the camera faster. (#99)
- Agents can have a rotation specified in the scenario config files (#209)
- Custom scenarios can be made with dictionaries as well as `json` files. See *Custom Scenario Configurations* (#275)
- Documented how to improve Holodeck performance. See *Improving Holodeck Performance* (#109)

6.3.2 Bug Fixes

- Fixed *`info()`* method (#182)
- Fixed command buffer not being reset after calling *`reset()`*. (#254)
- Fixed rain not being very visible on Linux (#235)
- Fixed teleport command not working on the Android (#209)
- Fixed RGBCamera intermittently returning a matrix of zeros after resetting (#271)
- Fixed `EXCEPTION_ACCESS_VIOLATION` on Windows after creating an environment (#270)
- Fixed *MazeWorld-FinishMazeSphere* task not going terminal when task was finished.
  - Added a post with a golden ball on top to the end of the maze, this is now the tasks's target

6.4 Holodeck 0.2.1

05/20/2019

This release of Holodeck is focused on polishing existing features and allowing worlds to be customized more.

This summer we are planning on adding much more content (worlds, agents, etc).

6.4.1 Highlights

- Added *Scenarios* to allow worlds to be more flexible and customizable
- Documentation has been greatly expanded

6.4.2 New Features

- Added expanded teleport functionality (#128)
- Add ticks per capture command for RGB Camera (#127)
- Add `__enter__` and `__exit__` methods to *`HolodeckEnvironment`* (#125)
- Add option to run headless on Linux (*`set_render_quality()`* on *`HolodeckEnvironment`*) (#135)
- Add ability to adjust rendering options (*`set_render_quality()`*) (#136)
- Add environment flag that allows state to be returned as copied object instead of reference (#151)
- Packages are not hard-coded on server, binaries are saved in version-specific folder to prevent crosstalk (#188)
- Sensors can be disabled to improve performance (#152)
- Add the ability to draw points, lines, arrows and boxes in the worlds (#144)
- Added new tasks for use with scenarios
- Added new scaled torque control scheme to the Android (#150)

6.4.3 Bug Fixes

- Fixed `mmap length is greater than filesize` error on startup (#115)
- Make all unit conversions on holodeck-engine side (#162)
- Fix multi-agent example (thanks bradyz!) (#118)
- Make sure *`reset()`* called before *`tick()`* and *`act()`* (#156)
- And many smaller bugs!

6.5 Holodeck 0.1.0

Initial public release.

CHAPTER

SEVEN

HOLODECK

Module containing high level interface for loading environments.

Classes:

GL_VERSION()	OpenGL Version enum.

Functions:

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

class holodeck.holodeck.**GL_VERSION**

OpenGL Version enum.

OPENGL3

The value for OpenGL3.

Type int

OPENGL4

The value for OpenGL4.

Type int

holodeck.holodeck.**make**(*scenario_name=''*, *scenario_cfg=None*, *gl_version=4*, *window_res=None*, *verbose=False*, *show_viewport=True*, *ticks_per_sec=30*, *copy_state=True*)

Creates a Holodeck 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. Defaults to 30.
- **copy_state** (bool, optional) – If the state should be copied or passed as a reference when returned. Defaults to True

Returns

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

Return type *HolodeckEnvironment*

CHAPTER

EIGHT

AGENTS

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

Definitions for different agents that can be controlled from Holodeck

Classes:

AgentDefinition(agent_name, agent_type[, ...])	Represents information needed to initialize agent.
AgentFactory()	Creates an agent object
AndroidAgent(client[, name])	An humanoid android agent.
ControlSchemes()	All allowed control schemes.
HandAgent(client[, name])	A floating hand agent.
HolodeckAgent(client[, name])	A learning agent in Holodeck
NavAgent(client[, name])	A humanoid character capable of intelligent navigation.
SphereAgent(client[, name])	
TurtleAgent(client[, name])	A simple turtle bot.
UavAgent(client[, name])	

class holodeck.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 HolodeckAgent to control, string or class reference.
- **sensors** (*SensorDefinition* or class type (if no duplicate sensors)) – A list of HolodeckSensors 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 holodeck.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** (*holodeck.holodeckclient.HolodeckClient*) – HolodeckClient agent is associated with
- **agent_def** (*AgentDefinition*) – Definition of the agent to instantiate

Returns:

class holodeck.agents.**AndroidAgent**(*client*, *name='DefaultAgent'*)

An humanoid android agent.

Can be controlled via torques supplied to its joints.

See *AndroidAgent* for more details.

Action Space:

94 dimensional vector of continuous values representing torques to be applied at each joint. The layout of joints can be found here:

There are 18 joints with 3 DOF, 10 with 2 DOF, and 20 with 1 DOF.

Inherits from *HolodeckAgent*.

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.
joint_ind(joint_name)	Gets the joint indices for a given name

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

(

static joint_ind(*joint_name*)

Gets the joint indices for a given name

Parameters **joint_name** (str) – Name of the joint to look up

Returns The index into the state array

Return type (int)

class holodeck.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 holodeck.agents.**HandAgent**(*client*, *name='DefaultAgent'*)

A floating hand agent.

Can be controlled via torques supplied to its joints and moved around in three dimensions.

See *HandAgent* for more details.

Action Space:

23 or 26 dimensional vector of continuous values representing torques to be applied at each joint. The layout of joints can be found here: *HandAgent Joints*.

Inherits from *HolodeckAgent*.

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.
joint_ind(joint_name)	Gets the joint indices for a given name

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

(

static joint_ind(*joint_name*)
Gets the joint indices for a given name

Parameters **joint_name** (`str`) – Name of the joint to look up

Returns The index into the state array

Return type (int)

class `holodeck.agents.`**HolodeckAgent**(*client*, *name='DefaultAgent'*)
A learning agent in Holodeck

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

Parameters

- **client** (*HolodeckClient*) – The HolodeckClient 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`, *HolodeckSensor*)) – A list of HolodeckSensors to read from this agent.

name
The name of the agent.

Type `str`

sensors
List of HolodeckSensors on this agent.

Type dict of (string, *HolodeckSensor*)

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 (*HolodeckSensor* or: list of *HolodeckSensor*): 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 (*HolodeckSensor* or: list of *HolodeckSensor*): 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 holodeck.agents.**NavAgent**(*client*, *name='DefaultAgent'*)
A humanoid character capable of intelligent navigation.

See *NavAgent* for more details.

Action Space:

Continuous control scheme of the form [x_target, y_target, z_target]. (see *Coordinate System*)

Inherits from *HolodeckAgent*.

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 holodeck.agents.**SphereAgent**(*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

(

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

See *TurtleAgent* for more details.

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 *HolodeckAgent*.

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 holodeck.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 Holodeck. An environment contains all elements required to communicate with a world binary or HolodeckCore editor.

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

Classes:

HolodeckEnvironment([agent_definitions, ...])	Proxy for communicating with a Holodeck world

class holodeck.environments.**HolodeckEnvironment**(*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=30*, *copy_state=True*, *scenario=None*)

Proxy for communicating with a Holodeck world

Instantiate this object using *holodeck.holodeck.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 holodeck. 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) – Number of frame ticks per unreal second. Defaults to 30.

- **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])	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
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_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 Holodeck.
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])	Supplies an action to the main agent and tells the environment to tick once.
tick([num_ticks])	Ticks the environment once.

Attributes:

action_space	Gives the action space for the main agent.

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.** –

draw_arrow(*start*, *end*, *color=None*, *thickness=10.0*)
Draws a debug arrow in the world

Parameters

- **start** (list of float) – The start [x, y, z] location of the line. (see *Coordinate System*)
- **end** (list of float) – The end [x, y, z] location of the arrow
- **color** (list) – [r, g, b] color value
- **thickness** (float) – thickness of the arrow

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

Parameters

- **center** (list of float) – The start [x, y, z] location 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
- **thickness** (float) – thickness of the lines

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

Parameters

- **start** (list of float) – The start [x, y, z] location of the line. (see *Coordinate System*)
- **end** (list of float) – The end [x, y, z] location of the line
- **color** (list`) – [r, g, b] color value
- **thickness** (float) – thickness of the line

draw_point(*loc*, *color=None*, *thickness=10.0*)
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
- **thickness** (float) – thickness of the point

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 obj)

Return type

(

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

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 [roll, pitch, yaw] rotation to give the camera (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 (tuple or dict): For single agent environment, returns the same as *step*.

For multi-agent environment, returns the same as *tick*.

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 Holodeck.

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, like the *Location Task*.

step (*action*, *ticks=1*)
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 wiht this action. If ticks > 1, this function returns the last state generated.

Returns

A 4tuple:

- **State: Dictionary from sensor enum** (see *HolodeckSensor*) to np.ndarray.
- Reward (float): Reward returned by the environment.
- Terminal: The bool terminal signal returned by the environment.
- Info: Any additional info, depending on the world. Defaults to None.

Return type (dict, float, bool, info)

tick (*num_ticks=1*)
Ticks the environment once. Normally used for multi-agent environments. :param num_ticks: Number of ticks to perform. Defaults to 1. :type num_ticks: int

Returns

A dictionary from agent name to its full state. The full state is another dictionary from `holodeck.sensors.Sensors` enum to np.ndarray, containing the sensors information for each sensor. The sensors always include the reward and terminal sensors.

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 holodeck.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 holodeck.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 holodeck.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 Holodeck backend. Most of these commands are just used internally by Holodeck, 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 Holodeck
RenderViewportCommand(render_viewport)	Enable or disable the viewport.
RotateSensorCommand(agent, sensor, rotation)	Rotate a sensor on the agent
SpawnAgentCommand(location, rotation, name, ...)	Spawn an agent in the world.
TeleportCameraCommand(location, rotation)	Move the viewport camera (agent follower)

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

Add a sensor to an agent

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

class holodeck.command.**Command**

Base class for Command objects.

Commands are used for IPC between the holodeck python bindings and holodeck 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.

continues on next page

Table 2 – continued from previous page

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

Parameters **client** (*HolodeckClient*) – 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 holodeck.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 holodeck.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 holodeck.command.**DebugDrawCommand**(*draw_type*, *start*, *end*, *color*, *thickness*)
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 of the object. (see *Coordinate System*)
- **end** (list of float) – The end [x, y, z] location of the object (not used for point, and extent for box)
- **color** (list of float) – [r, g, b] values for the color
- **thickness** (float) – thickness of the line/object

class holodeck.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 holodeck.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 holodeck.command.**RenderQualityCommand**(*render_quality*)

Adjust the rendering quality of Holodeck

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

class holodeck.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 holodeck.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 holodeck.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 holodeck.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

HOLODECK CLIENT

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

Classes:

HolodeckClient([uuid, should_timeout])	HolodeckClient for controlling a shared memory session.

class holodeck.holodeckclient.**HolodeckClient**(*uuid=''*, *should_timeout=False*)

HolodeckClient 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 "".
- **should_timeout** (boolean, optional) – If the client should time out after 5s waiting for the engine

Methods:

acquire()	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()

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 Holodeck

Functions:

available_packages()	Returns a list of package names available for the current version of Holodeck
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])	Installs a holodeck 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 holodeck, other than the running version.
remove(package_name)	Removes a holodeck package.
remove_all_packages()	Removes all holodeck 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.

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

Returns (list of str): List of package names

holodeck.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

holodeck.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

holodeck.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

holodeck.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

holodeck.packagemanager.**install**(*package_name*, *url=None*)
Installs a holodeck package.

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

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

Returns List of all the currently installed packages

Return type list of str

holodeck.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

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

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

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

DO NOT USE WITH HOLODECKPATH

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

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

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

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

holodeck.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

holodeck.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:

AbuseSensor(client[, agent_name, ...])	Returns True if the agent has been abused.
AvoidTask(client[, agent_name, agent_type, ...])	
BallLocationSensor(client[, agent_name, ...])	For the CupGame task, returns which cup the ball is underneath.
CleanUpTask(client[, agent_name, ...])	
CollisionSensor(client[, agent_name, ...])	Returns true if the agent is colliding with anything (including the ground).
CupGameTask(client[, agent_name, ...])	
DistanceTask(client[, agent_name, ...])	
FollowTask(client[, agent_name, agent_type, ...])	
HolodeckSensor(client[, agent_name, ...])	Base class for a sensor
IMUSensor(client[, agent_name, agent_type, ...])	Inertial Measurement Unit sensor.
JointRotationSensor(client, agent_name, ...)	Returns the state of the *AndroidAgent*'s or the *HandAgent*'s joints.
LocationSensor(client[, agent_name, ...])	Gets the location of the agent in the world.
LocationTask(client[, agent_name, ...])	
OrientationSensor(client[, agent_name, ...])	Gets the forward, right, and up vector for the agent.
PressureSensor(client, agent_name, agent_type)	For each joint on the *AndroidAgent* or the *HandAgent*, returns the pressure on the joint.
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.
RelativeSkeletalPositionSensor(client, ...)	Gets the position of each bone in a skeletal mesh as a quaternion.
RotationSensor(client[, agent_name, ...])	Gets the rotation of the agent in the world.
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 HolodeckSensor object.
VelocitySensor(client[, agent_name, ...])	Returns the x, y, and z velocity of the agent.
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 holodeck.sensors.**AbuseSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Returns True if the agent has been abused. Abuse is calculated differently for different agents. The Sphere and Hand agent cannot be abused. The Uav, Android, and Turtle agents can be abused by experiencing high levels of acceleration. The Uav is abused when its blades collide with another object, and the Turtle agent is abused when it's flipped over.

Configuration

- `AccelerationLimit`: Maximum acceleration the agent can endure before being considered abused. The default depends on the agent, usually around 300 m/s^2.

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 holodeck.sensors.**AvoidTask**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

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 holodeck.sensors.**BallLocationSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

For the CupGame task, returns which cup the ball is underneath.

The cups are numbered 0-2, from the agents perspective, left to right. As soon as a swap begins, the number returned by this sensor is updated to the balls new position after the swap ends.

Only works in the CupGame world.

Attributes:

dtype	The type of data in the sensor

property dtype
The type of data in the sensor

Returns Type of sensor data

Return type numpy dtype

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

Attributes:

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

Methods:

start_task(num_trash[, use_table])	Spawn trash around the trash can.

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

start_task(*num_trash*, *use_table=False*)
Spawn trash around the trash can. Do not call if the config file contains a clean up task configuration block.

Parameters

- **num_trash** (int) – Amount of trash to spawn
- **use_table** (bool, optional) – If True a table will spawn next to the trash can, all trash will be on the table, and the trash can lid will be absent. This makes the task significantly easier. If False, all trash will spawn on the ground. Defaults to False.

class holodeck.sensors.**CollisionSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)
Returns true if the agent is colliding with anything (including the ground).

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 holodeck.sensors.**CupGameTask**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Attributes:

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

Methods:

start_game(num_shuffles[, speed, seed])	Start the cup game and set its configuration.

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

start_game(*num_shuffles*, *speed=3*, *seed=None*)
Start the cup game and set its configuration. Do not call if the config file contains a cup task configuration block, as it will override the configuration and cause undefined behavior.

Parameters

- **num_shuffles** (int) – Number of shuffles
- **speed** (int) – Speed of the shuffle. Works best between 1-10
- **seed** (int) – Seed to rotate the cups the same way every time. If none is given, a seed will not be used.

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

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 holodeck.sensors.**FollowTask**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

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 holodeck.sensors.**HolodeckSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Base class for a sensor

Parameters

- **client** (*HolodeckClient*) – 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 holodeck.sensors.**IMUSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)
Inertial Measurement Unit sensor.

Returns a 2D numpy array of

::`

[[acceleration_x, acceleration_y, acceleration_z], [velocity_roll, velocity_pitch, velocity_yaw]]

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 holodeck.sensors.**JointRotationSensor**(*client*, *agent_name*, *agent_type*, *name='RGBCamera'*, *config=None*)
Returns the state of the `AndroidAgent`'s or the `HandAgent`'s joints.

See *Android Joints* or *HandAgent Joints* for the indexes into this vector.

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 holodeck.sensors.**LocationSensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

Gets the location of the agent in the world.

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

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 holodeck.sensors.**LocationTask**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)

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 holodeck.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, forward_y, forward_z],
  [right_x,   right_y,   right_z  ],
  [up_x,      up_y,      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 holodeck.sensors.**PressureSensor**(*client*, *agent_name*, *agent_type*, *name='RGBCamera'*, *config=None*)

For each joint on the *AndroidAgent* or the *HandAgent*, returns the pressure on the joint.

For each joint, returns [x_loc, y_loc, z_loc, force], in the order the joints are listed in *Android Joints* or *HandAgent Joints*.

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 holodeck.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

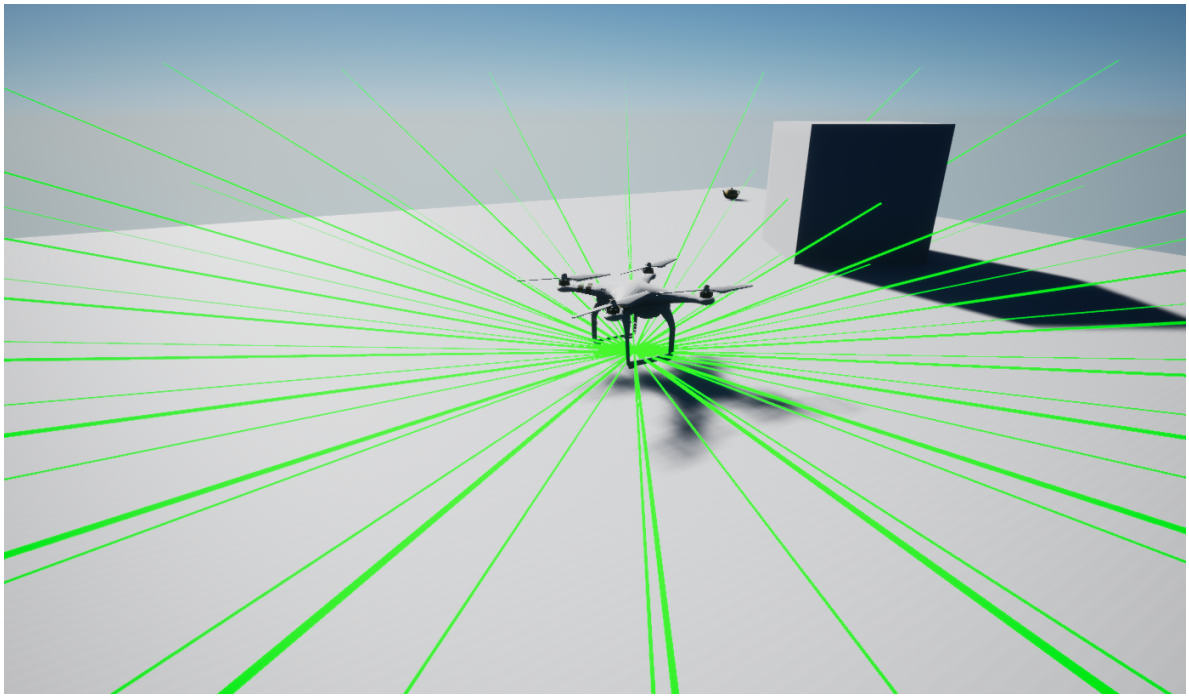
set_ticks_per_capture(*ticks_per_capture*)
Sets this RGBCamera to capture a new frame every ticks_per_capture.

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 holodeck.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 holodeck.sensors.**RelativeSkeletalPositionSensor**(*client*, *agent_name*, *agent_type*, *name='RGBCamera'*, *config=None*)

Gets the position of each bone in a skeletal mesh as a quaternion.

Returns a numpy array with four entries for each bone (see *AndroidAgent Bones* or *HandAgent Bones* for the order used)

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

Gets the rotation of the agent in the world.

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 `holodeck.sensors.`**SensorDefinition**(*agent_name*, *agent_type*, *sensor_name*, *sensor_type*, *socket=''*, *location=(0, 0, 0)*, *rotation=(0, 0, 0)*, *config=None*, *existing=False*)

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 *HolodeckSensor*) – 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 holodeck.sensors.**SensorFactory**
Given a sensor definition, constructs the appropriate HolodeckSensor 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 holodeck.sensors.**VelocitySensor**(*client*, *agent_name=None*, *agent_type=None*, *name='DefaultSensor'*, *config=None*)
Returns the x, y, and z velocity of the agent.

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 holodeck.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 holodeck.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

SHARED MEMORY

Shared memory with memory mapping

Classes:

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

class holodeck.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

SIXTEEN

UTIL

Helpful Utilities

Functions:

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

holodeck.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)

holodeck.util.**get_holodeck_path**()
Gets the path of the holodeck environment

Returns path to the current holodeck environment

Return type (str)

holodeck.util.**get_holodeck_version**()
Gets the current version of holodeck

Returns the current version

Return type (str)

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

Returns Linux or Windows. Throws NotImplementedError for other systems.

Return type str

holodeck.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

SEVENTEEN

EXCEPTIONS

Holodeck Exceptions

Exceptions:

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

exception holodeck.exceptions.**HolodeckConfigurationException**
The user provided an invalid configuration for Holodeck

exception holodeck.exceptions.**HolodeckException**
Base class for a generic exception in Holodeck.

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

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

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

CHAPTER

EIGHTEEN

WEATHER CONTROLLER

Weather/time controller for environments

Classes:

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

class holodeck.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

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

E

F

G

H

I

J

L

T

U

V

W